1995

Parallel algorithms and permutation

Lap K. Mui
San Jose State University

PARALLEL ALGORITHMS AND PERMUTATION

A Thesis

Presented to

The Faculty of the Department of Mathematics and Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

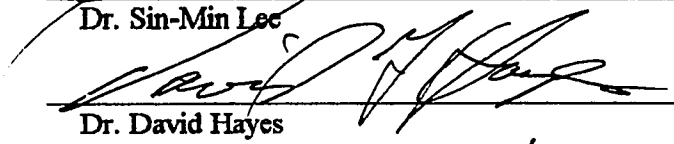
by

Lap K. Mui

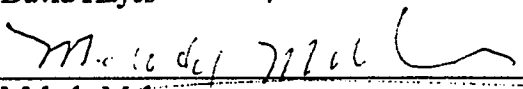
August 1995

APPROVED FOR THE DEPARTMENT OF
MATHEMATICS AND COMPUTER SCIENCE

Dr. Sin-Min Lee



Dr. David Hayes



Dr. Melody Moh

APPROVED FOR THE UNIVERSITY

ABSTRACT

PARALLEL ALGORITHMS AND PERMUTATION

by

Lap K. Mui

This thesis examines the problem of parallel algorithm design. Through research into parallel architectures and algorithms, this thesis demonstrates how to solve a computer science problem with a parallel solution. The goal of the thesis is to design a new parallel algorithm to generate permutations in lexicographic order. The parallel architecture being used in the algorithm should be simple in order to facilitate VLSI implementation. Moreover, the algorithm should be time efficient and cost optimal.

After studies of existing permutation algorithms, this thesis presents a new parallel algorithm that generates all the m-permutations of N objects in lexicographic order. It runs on a linear processor array of m processing elements and has a running time of $O({}^{N}P_{m})$. The delay time between successive permutations is constant. Therefore, the algorithm is time efficient and cost optimal and satisfies every aspect of our original goal.

ACKNOWLEDGMENTS

I would like to thank my committee members. I would like to thank Dr. Sin-Min Lee, my thesis advisor, for his guidance and support. I would like to thank Dr. David Hayes and Dr. Melody Moh for spending their time reading my thesis and giving me valuable opinions.

Table of Contents:

List of Figures:

Figure

Figure

Chapter 1. Introduction

As computing technology improves, the demand for fast and high performance computers is always increasing. There are always some applications, such as weather forecasting, simulation and artificial intelligence, which demand a computer with performance that has exceeded the current capability. In other words, we are facing an unlimited demand to improve the performance of the computer.

One way to increase the speed of the computer is to increase the physical speed of the switching devices by making the circuit elements smaller and faster. However, since propagation speed can never exceed the speed of light, this improvement has a limit. To satisfy the unlimited demand for improvement, some alternatives have to be found.

In everyday situations, when there is a task that cannot be completed by one individual, the most natural way to tackle this problem is to ask for help, i.e. get more than one individual to work on the problem simultaneously in a coordinated manner. That is the idea of parallel computing, and is also the reason why parallel processing has been under a lot of intensive research in recently years.

The goal of this thesis is to examine parallel algorithm design through the study of parallel architectures and algorithms, and by developing a new parallel algorithm to generate permutations in lexicographic order. This thesis can be divided into two parts. The first part of the thesis will cover parallel architectures and parallel sorting algorithms; this will serve as a preparation for designing an algorithm in the latter part of the thesis. The second part of the thesis will include research into some existing permutation algorithms and eventually presents a new parallel algorithm to generate permutations lexicographically.

Chapter 2. Parallel Architectures

2. 1 Introduction

Before considering parallel architectures, we need to answer the following questions.

How can we perform parallel computations on a computer?

The answer is obvious. The computer needs to have more than one processing element. It can be physical - multiple processors, or it can be conceptual - pipelining or context switching. These processing elements need to be organized in such a way that they can perform concurrent tasks in a efficient and coordinated fashion.

What is a parallel computer? How is it different than a sequential computer?

Throughout the history of computers, the definitions for parallel computation kept on changing. Forty years ago, arithmetic operation that process words instead of bits was considered to be a form of parallel computation. In more recent years, a few low-level parallel mechanisms have been introduced into computers, such as auxiliary processors, multiple functional units, time sharing and instruction pipelining. Almost all of today's computers are equipped with these features, but they are still being considered to be sequential machines.

The truth is there is no universal accepted definition to distinguish a parallel machine from a sequential machine. In this study, the following definition of parallel computer will be used:

A parallel system is a system consists of tightly or loosely coupled processing elements that are coordinated to accomplish a common task with a concurrent solution.

What are the distinctions between different parallel architectures?

Most sequential machines fit into a clearly defined architectural model: a combination of the von Neumann Machine with addition of some recent developments such as virtual memory and concurrent I/O.

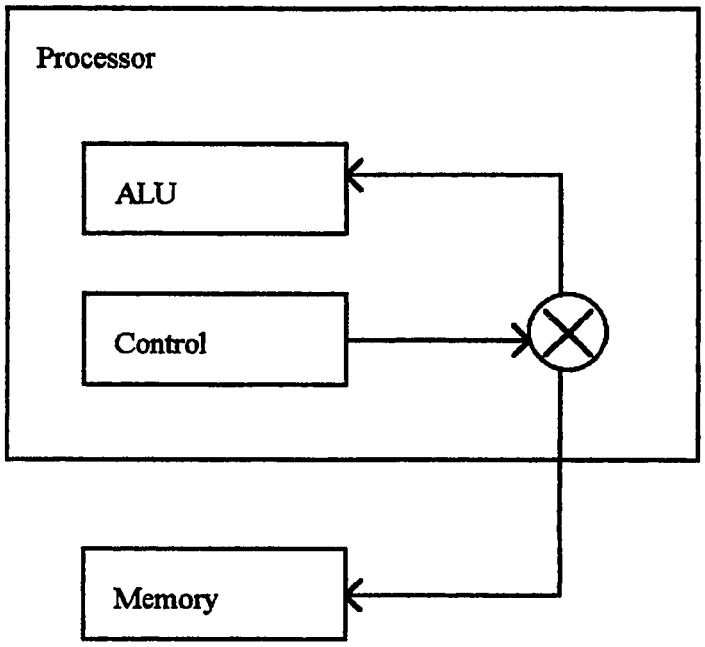


Figure 2.1 : Simple "von Neumann" computation model

The von Neumann model (Figure 2. 1) consists of the following main features :

1. A processor that performs instructions.
2. A memory that stores both the instructions and the data of a program.
3. A control unit that fetches instructions from the memory and transfers data between the processor and memory.

Unfortunately, parallel architectures do not have this kind of clear definition. Therefore, a more detailed classification scheme which will characterize all computer architecture (including parallel architectures), the Fynn's model, is necessary.

2. 2 Flynn's Classification

		number of data streams	
		single	multiple
number of instruction streams	single	SISD von Neumann	SIMD vector, array processors
	multiple	MISD pipeline ?	MIMD multiple microprocessors

Figure 2.2 : Flynn's classification.

Flynn's model is one of the most widely used classification schemes (Figure 2.2). In the Flynn's model, architectures are classified by their numbers of instruction streams and data streams. If we only consider whether the numbers of instruction streams and data streams are single or multiple, this classification consists of four classes of computational models: SISD, SIMD, MISD and MIMD.

2. 2. 1. Single Instruction Single Data (SISD)

This is the classical uniprocessor architecture, even though it may include some parallel mechanism.

The SISD model (Figure 2.3) has a single control unit. The control unit feeds the processor instructions from the memory module, while data being transferred between the processor and the memory module. It is a typical von Neumann machine.

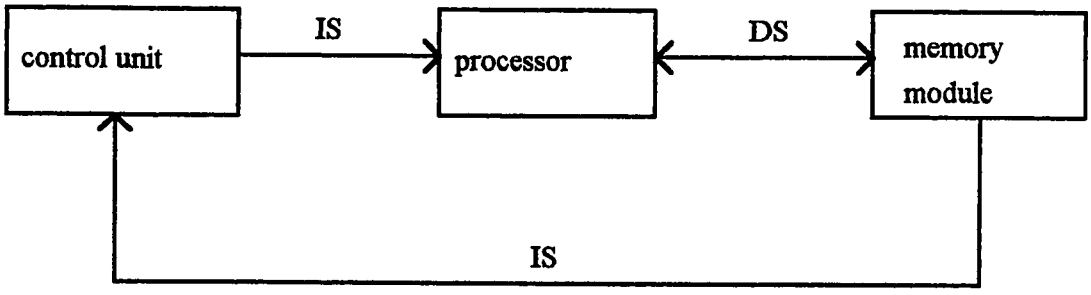


Figure 2.3 : SISD model.

The performance of this model depends on the execution speed of the processor and the transmission speed of data. To increase the execution speed of the processor, we can increase the speed of the switching devices or introduce some parallel computing mechanisms such as instruction pipelining into the processor. To increase the transmission speed of data, we can utilize some advance memory hierarchy such as memory interleaving or caching.

However, these improvements all have their limits. Pipelining has to deal with pipeline hazards such as data dependence. Memory interleaving is more useful in data-parallel programs where a small set of instructions are executed on a large set of data. On the other hand, memory caching is limited by the pattern of memory accesses and the limits of hardware technology improvement.

An alternative way to increase the execution speed and transmission speed of the computer is to use multiple processors and organize the memory modules in some special fashion. In other words, introduction of parallel computing is essential.

2. 2. 2. Multiple Instruction Single Data (MISD)

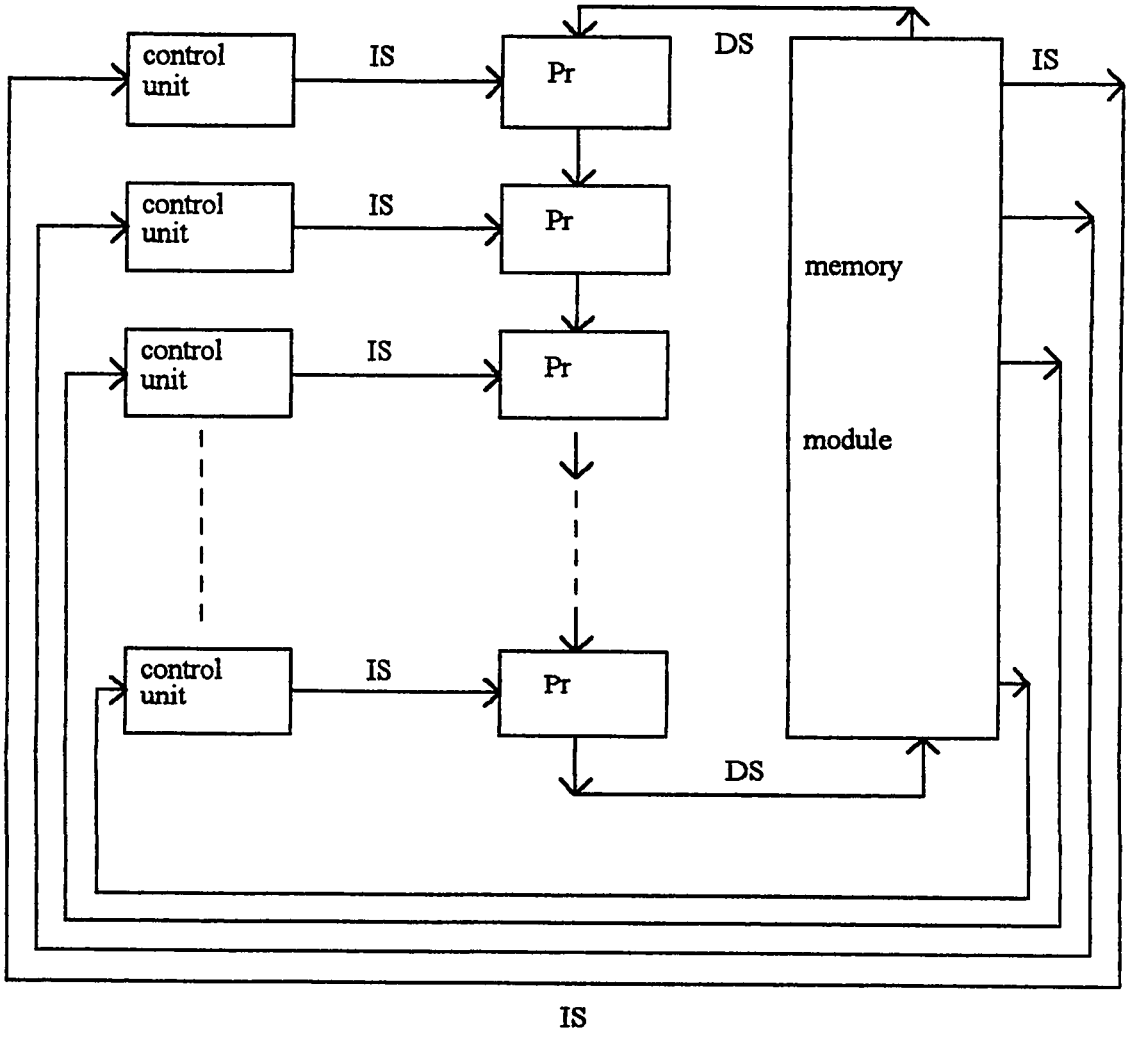


Figure 2.4: MISD model.

In the MISD model (Figure 2.4), separate control units supervise their own processors. Each processor works on the same stream of data items with different instructions. The MISD model has never been implemented and is considered to be impractical.

2. 2. 3. Single Instruction Multiple Data (SIMD)

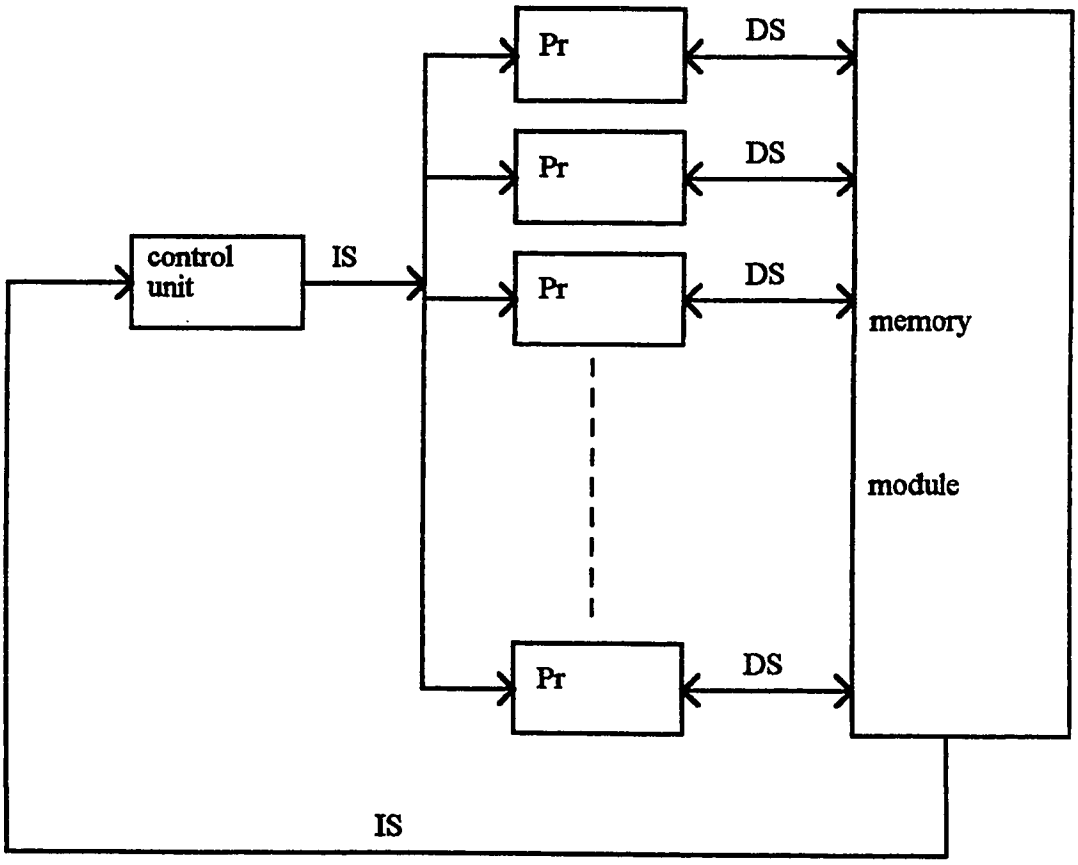


Figure 2.5: SIMD model.

The SIMD model (Figure 2.5) is supervised by a common control unit, each processor executes the same instruction simultaneously on its own set of data.

It is faster than the SISD model because the SIMD model avoids a separate instruction fetch for each data item, and it executes the instructions in parallel.

Some commercial SIMD machines are the Illiac IV, CM-2 and MP-1.

2. 2. 4. Multiple Instruction Multiple Data (MIMD)

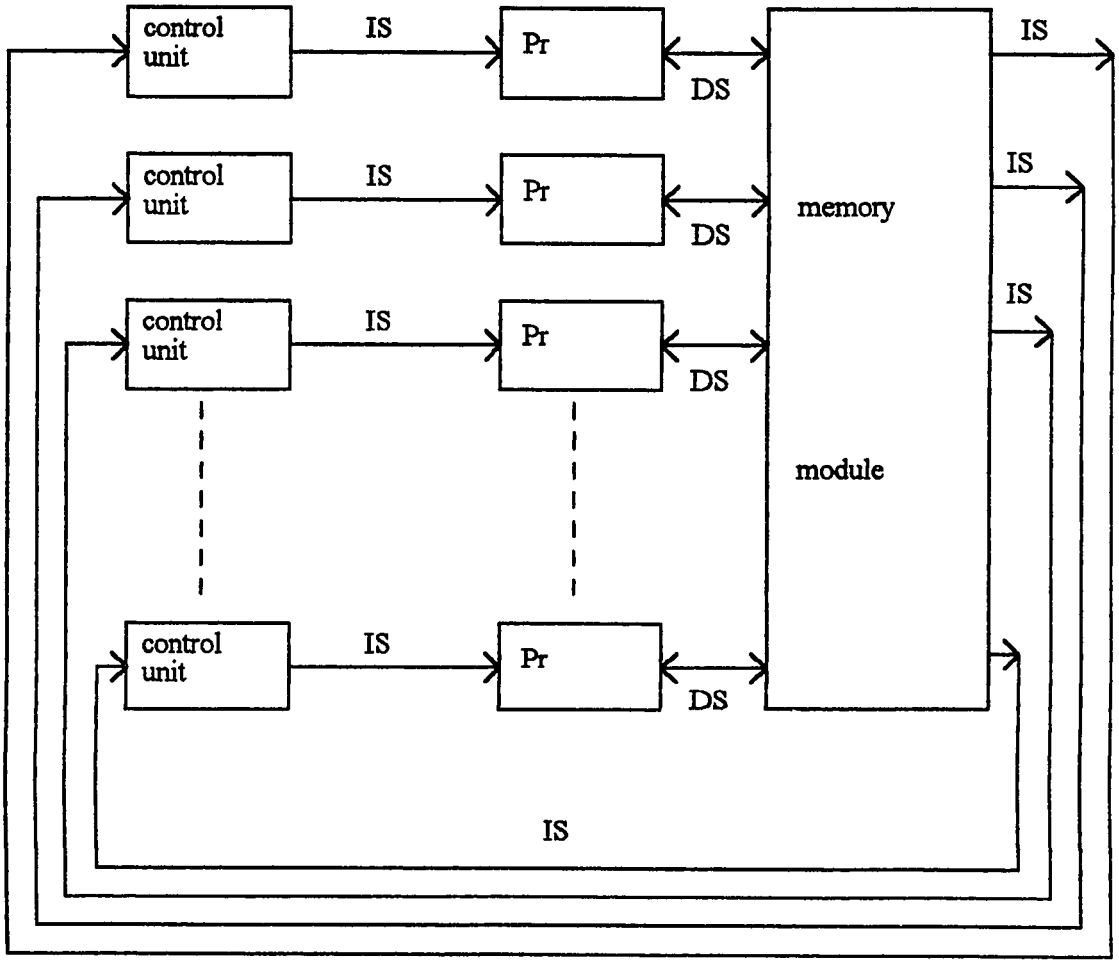


Figure 2.6: MIMD model.

In the MIMD model (Figure 2.6), several processors are controlled by different control units. Simultaneously, those processors execute different instructions on their own set of data items.

Examples of MIMD computers includes the nCUBE 2, iPSC and Symmetry.

2. 2. 5. Comparisons of SIMD and MIMD Models

Obviously, SIMD and MIMD are the two models that are suitable for implementing parallel architectures. Now, let's examine the differences between the two models.

SIMD computers have only one global control unit and each processor executes a broadcast instruction. Therefore, SIMD computers offer simpler synchronization. SIMD processors can be smaller and more numerous because they doesn't need to store their own programs. As a result, SIMD computers generally require less hardware. On the other hand, MIMD computers are more general, and capable of implementing a broader range of applications. Nevertheless, SIMD system can emulate a MIMD system by using an interpreter and vise versa.

On the surface, it may seem that MIMD processors are more expensive than SIMD processors because MIMD processors are more complex. However, each processor in a MIMD system can be a general-purpose microcomputer, such as in the case of a massive parallel system. On the other hand, processors in a SIMD computer must be specially designed and therefore a SIMD computer may be more expensive than a MIMD computer.

It may also seem that MIMD computers are strictly more powerful than SIMD computers with the same number of processors since each processor in a MIMD computer is more powerful. However, SIMD computers are more suitable to solve problems with regular structures. In fact, MIMD computers are not suitable for those problems

because they required the MIMD processors to be precisely synchronized to implement certain algorithms, and this synchronization cost can be significant.

In reality, pure MIMD computers have no hardware feature to guarantee synchronization of processors. In general, we can't load multiple copies of a program into all of the processors and start executing the programs at the same time and hope that they will execute the programs at the same rate. In fact, many such computers have hardware that tends to destroy synchronization once it has been achieved. For example, the way the Sequent Symmetry series machines access memories generally causes processors to run at different rates even if they are synchronized at some time. Many Sequent machines even have processors that run at different clock rates [Tal, Navathe, Graham [22]].

As a result, this asynchronous characteristic of MIMD machines becomes a problem and leads to the development of a new category of parallel computer that is a hybrid between SIMD and MIMD machines.

2. 2. 6. Synchronous-Asynchronous Multiple Data (SAMD)

The SAMD model is a hybrid of the SIMD and the MIMD models. Essentially, it is a MIMD computer with hardware features that allows:

- Precise synchronization of processors to be easily achieved.
- Once it has been achieved, synchronization of processors can be maintained with little or no overhead.

The SAMD model is different from the pure MIMD machines such that the hardware maintains a uniform "heartbeat" throughout the machine. Therefore, when the same program is run on all processors, and all copies of the program are started at the same time, it is possible that the execution of all copies to be kept in lock-steps with essentially

no overhead. Such computers allow efficient execution of both MIMD and SIMD programs.

An example of a SAMD machine is the CM-5.

In the study of parallel architectures, SIMD and MIMD are the two models that we are interested in. Section 2.3 and 2.4 will cover parallel architectures in SIMD and MIMD model, respectively.

2. 3 SIMD Architectures

2. 3. 1. Pipeline Architectures

The name pipeline came from the analogy with a typical industry assembly line in which the work to be done is broken into small pieces and then being performed in a overlapping manner.

Pipelines are classified into two primary categories: arithmetic pipelines and instruction pipelines. Arithmetic pipelines are used to implement complex arithmetic operations such as floating point or vector operations. In an instruction pipeline, an instruction is partitioned into several parts that can be executed in an overlapping fashion with parts from the other instructions.

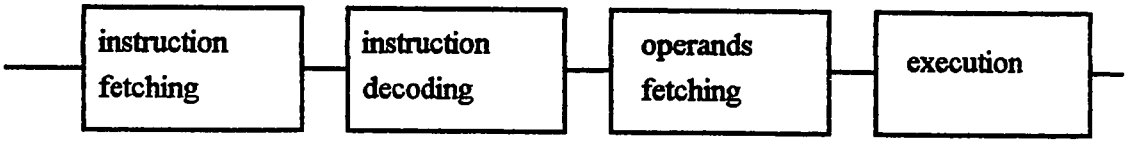


Figure 2.7: Instruction pipeline

Figure 2.7 is an example of a typical instruction pipeline. An instruction is broken into four pipeline stages: instruction fetch, instruction decode, operands fetch, and execution. Each instruction is executed in an overlapping fashion (Figure 2.8).

For the pipeline in Figure 2.8, ideally the instruction cycle can be reduced to one-fourth of the original cycle. Therefore, the speedup will be equal to the number of pipeline stages. However, in reality, such a speedup is impossible to achieve because of pipeline hazards such as conditional branch, data dependency and structural conflict.

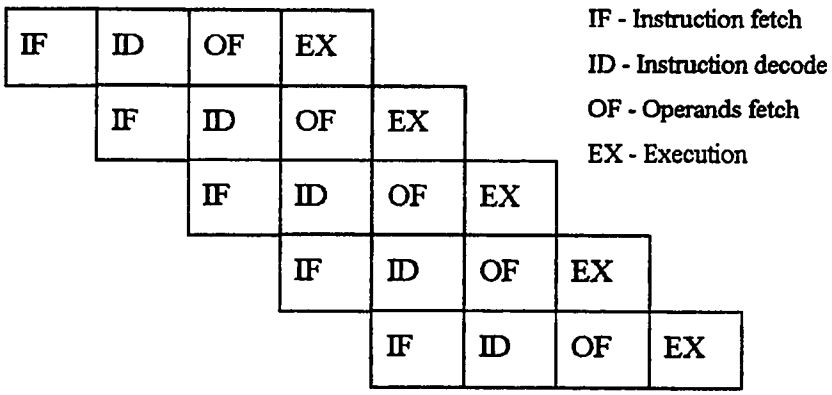


Figure 2.8: Pipeline timing diagram.

2. 3. 2. Array Processor Architectures

An array processor (Figure 2.9) is a regularly connected array of processing elements. The processing elements operate simultaneously and synchronously under the control of a common control unit. The control unit broadcasts a single instruction to be executed by each of the processor on its own set of data items. Each individual processor may have its own local memory and communicates with the others through an interconnection network. Or all the individual processors can access a common memory module through a network.

The array processors are suitable for solving very structured problems, such as problems involving vector computations.

However, to fully take advantage of the parallelism in this structure, the programmers must be aware of the computer's architecture when designing the algorithms. Therefore, this kind of processor is usually used in scientific and special applications.

An example of a computer with array processors is the Illiac IV.

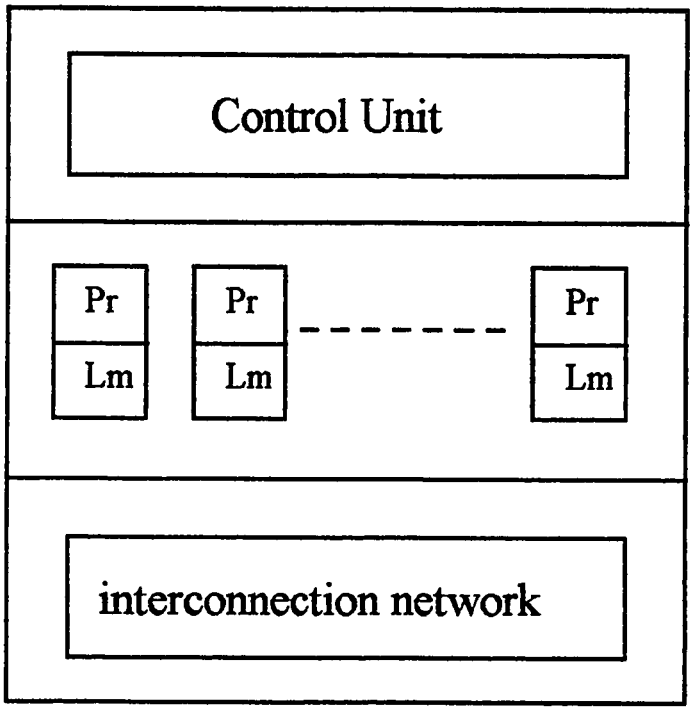


Figure 2.9: Array Processor.

2. 3. 3. Fine Grained Array Processor

An extension to the array processors architecture is the fine grained array processor. The most famous computer equipped with the fine grained array processor is the connection machine.

The connection machine is a fine-grained machine which consists of tens of thousands of simple processing elements. Each of the processing elements has a small local memory and a one bit wide arithmetic logic unit.

The array processor is connected to a front-end typical von Neumann machine (usually a Sun workstation) where the instructions to be executed by the processing elements are originated. A high speed I/O system allows efficient data transfers between the parallel processing units and I/O devices, such as frame buffers and parallel disk drives.

The front end machine can exchange data with the processing elements in three ways [Quinn [14]]:

1. It can broadcast a single value to all of the processing elements with the instruction broadcast bus.
2. Through global combining, it can obtain the sum, maximum, global OR, ... , etc., of one value from each processing element.
3. By using the scalar memory bus, it can read or write 32 bit values stored in any group of 32 processing elements.

The front end machine issues parallel processing instructions to the sequencer, which interprets each instruction and generates a series of nano-instructions. It broadcasts the nano-instructions over an instruction bus to the processing elements, which execute them.

In the CM-2, sixteen processors are connected in a grid. The processor groups are interconnected in the pattern of a Boolean n-cube (a 12-dimensional hypercube).

This architecture can be implemented with the virtual processor concept and a general purpose communication system. The communication system allows the processors to communicate with each other by sending a message to other processors' local memories. The virtual processor concept means a controller can be placed between the front-end

machine and the array processor, where the controller will control the array processor to adapt to different size problems and allow the number of processors in the system to be expanded. Therefore, the connection machine is scalable, processors can be added to the system without changing the programs. The connection machine works efficiently for massive data parallelism applications.

2. 3. 4. Associative Array Processors

Associative array processors are SIMD array processors built around associative memory. Associative memory is in a sense an active memory. Its storage cells are provided with special logical capabilities so that simple arithmetic and logic operations can be performed in the memory cells themselves. That allows searches and comparisons to be made in parallel. Moreover, the data stored in an associative memory is accessed by its content instead of its address. All these characteristics allow large amounts of information to be accessed very quickly. Therefore, in the area of information processing, associative array processors are very effective, particularly in applications that involved massive searches of a very large database such as weather forecasting computation and signal processing.

2. 3. 5. Systolic Arrays

Systolic architectures are typically large, regular arrays of simple processors. These processors operate in parallel, passing data between themselves continuously and synchronously in a fixed, regular pattern. One way to emphasize the characteristics of the systolic concept is by comparing it to the von Neumann approach of instruction execution.

In the von Neumann approach (Figure 2.10a), an instruction is fetched from the memory, decoded, and executed after the operands are fetched. Then, an operand will be yielded and stored into the memory again.

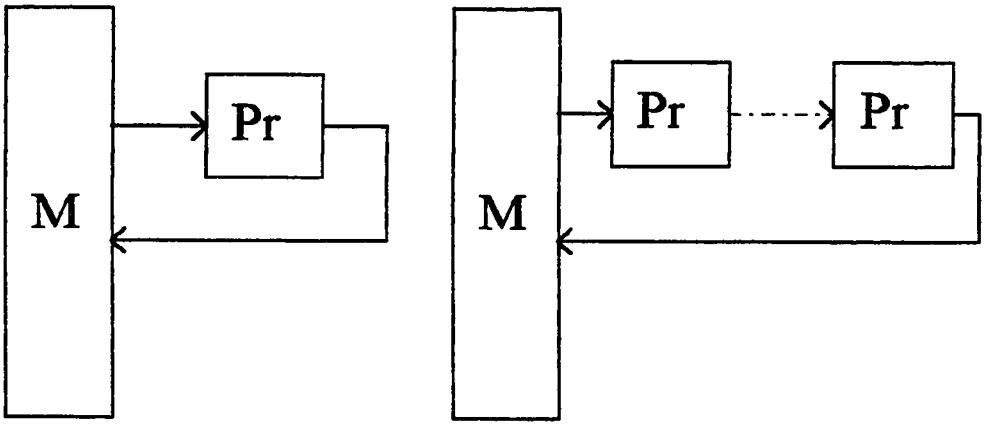


Figure 2.10a: von Neumann approach. Figure 2.10b: systolic approach.

In the systolic approach (Figure 2.10b), data is being pumped steadily through the array of cells without any memory reference. This significantly reduces the time complexity of the machine.

Planar systolic arrays with n^2 processors can often implement $O(n^3)$ algorithm in O(n) time [Tal, Navathe, Graham [22]]. They are able to do that because of the flow of data through the interconnection network is fixed, so there is no control overhead. Since very few processors are left idle, the operations are optimized.

As a result, systolic arrays are widely used as optimal solutions to many practical computational problems such as signal processing and matrix computation.

Systolic architectures favor the use of a few simple identical processing elements interconnected in a regular pattern. In consequence, it yields a short communication path and an economical VLSI design and implementation.

Systolic arrays have some drawbacks, such as their inefficient handling of complex data structures and non-scalar operands. Moreover, only a narrow set of problems can be mapped onto a particular systolic array.

2. 4 MIMD Architectures

2. 4. 1. Multiprocessor Architecture

A multiprocessor computer is one which employs two or more independent processors. Each processor operates under a combined control unit and they share memory modules, I/O channels, and an interconnection network.

The interconnection network is very important in determining the performance of the system. As a result, we can use the interconnection network to classify the multiprocessor architecture into three main categories: time shared buses, crossbar switching systems, and multistage networks.

Bus-oriented Systems

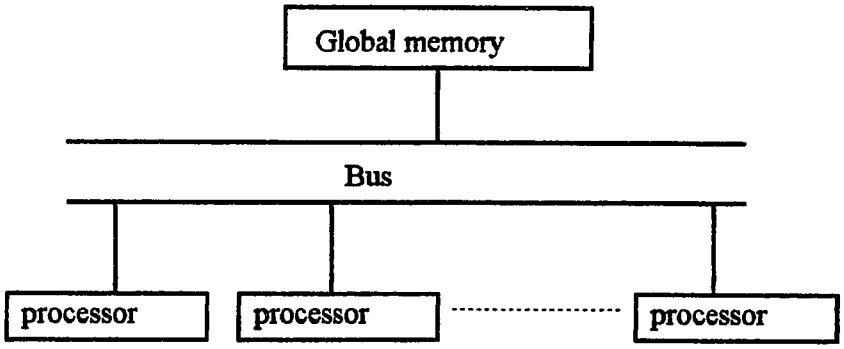


Figure 2.11: A single bus system.

Bus oriented systems contain one or more buses which connect all the processors and memory modules. A single-bus system (Figure 2.11) is the simplest and least expensive. It is extremely scalable. However, bus contentions may occur. Since there is only one bus, a bus failure will be catastrophic to the system, the whole system will be shut down.

Multibus systems overcome these problems. However, extra logic is needed and that increases the complexity of the system. Furthermore, for each component of the system, multiport capability is required to accommodate multiple buses.

Another way to alleviate the problem of bus contentions is to use a cache memory between the bus and the processors. It will reduce the number of accesses to the global memory. However, problem of cache coherence may occur when a processor modifies a shared variable in its cache and thus different processors will have different values for this variable.

Crossbar Switching Network

The crossbar switching network (Figure 2.12) is the optimal solution for the contention problem. In a crossbar switching system, all the components are physically interconnected by crosspoint switches. As long as no two accesses involve the same components at the same time, no contention will occur.

The crossbar system is characterized by its simple protocols and high hardware complexity. However, since each component has to be connected to all other components, for a N component system, it requires N^2 switching units. Consequently, crossbar switching networks are not very scalable in terms of cost. They are very expensive and impractical for a system with a large number of components.

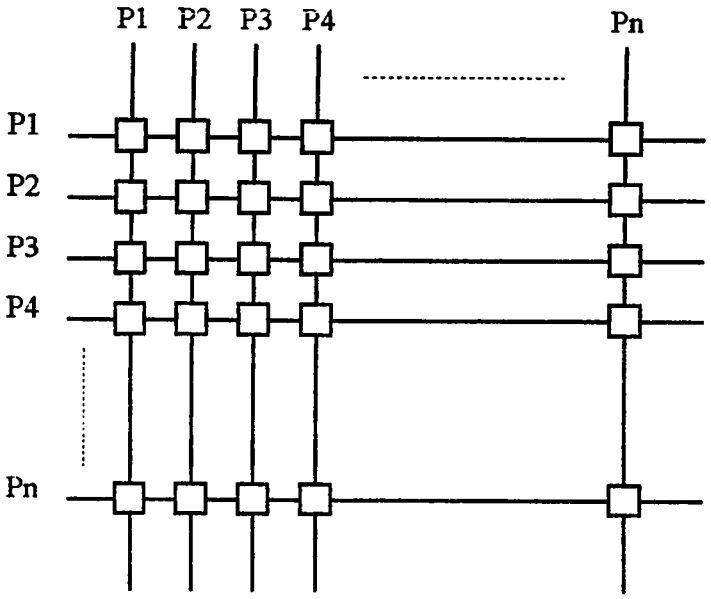


Figure 2.12: A crossbar switching network

Multistage Interconnection Network

A multistage network is a more practical approach than the crossbar switching idea. Components are connected by switches with a certain kind of interconnection. And they are being arranged in stages.

A switching network connecting N inputs to M outputs is called an NxM switching network. One of the basic requirement for the multistage network is each processor must be connected to all other processors.

Figure 2.13 is a three stage network. It has the property that all the processors are connected. A network that allows two processors to communicate with each other as long as they are not currently occupied is called a non-blocking network. Figure 2.13 is a non-blocking networking.

Parallel computers that based on a multistage network include the BBN Butterfly and the IBM RP-3.

More discussion of the multistage network can be found in [Kuman, Grama, Gupta & Karypis [10]].

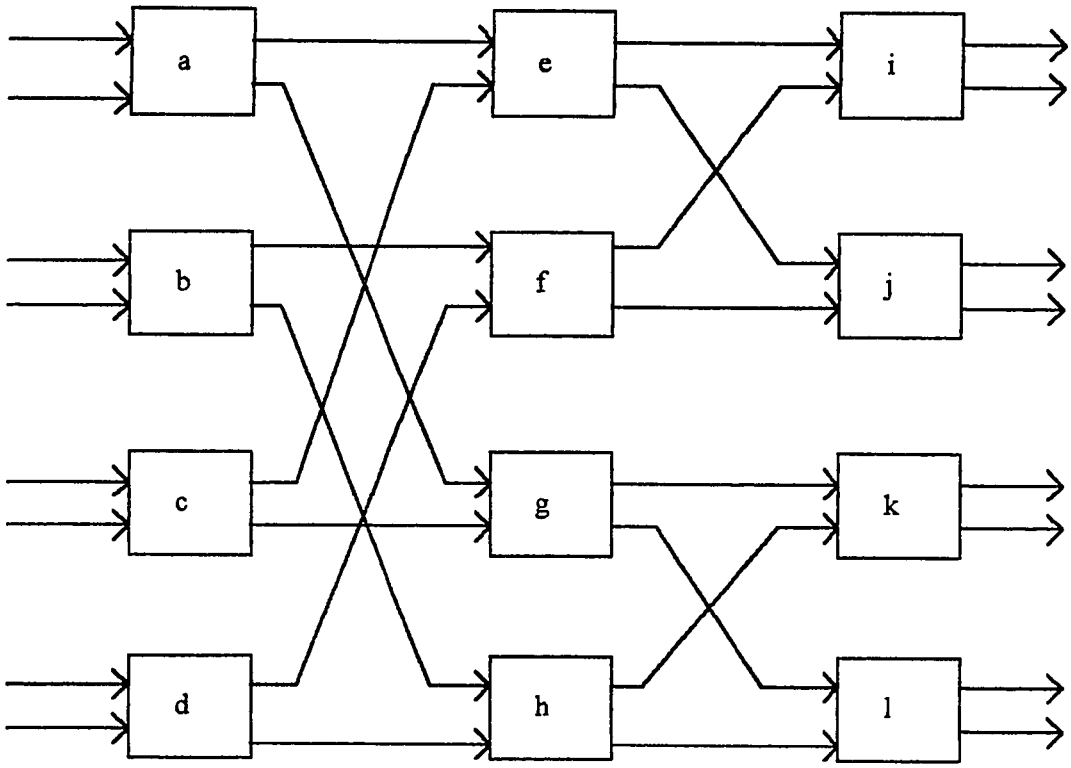


Figure 2.13: Three stage switching network

2. 4. 2. Data Flow Architecture

In the data flow model, all the instructions are considered as independent entities. Each instruction can be executed as an independent concurrent action whenever it has its

input data. This architecture tries to eliminate the bottleneck caused by the steady stream of data exchanged by the processors and the memories each time an instruction is executed. The data flow model of computation is based on two principles:

i) All operations are executed when and only when all the required operands are available.

ii) All operations are functions; there are no side effects.

The first principle distributes control to the level of operations, where an instruction can be executed at any time its operands are available. The second principle implies that enabled operations can be executed in any order since there are no side effects.

Figure 2.14 is an example data flow graph for the computation of $Z = (X+Y)*(X-Y)$. In this example, the data flow model allows three operations to be computed in two computation cycles where operation X+Y and X-Y can be computed simultaneously.

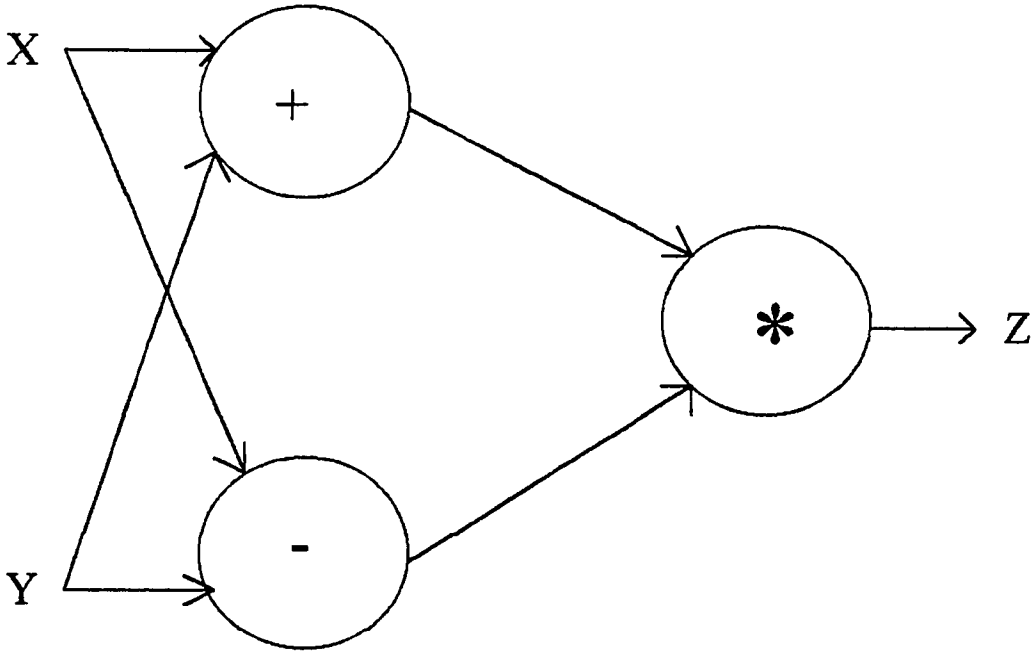


Figure 2.14: Data flow graph for Z = (X+Y) * (X-Y)

The data flow concept is very attractive because control is distributed out to the level of operations on scalar operands. Furthermore, parallelism in execution flows naturally as operands become available, thus following an algorithm's natural parallelism. Unfortunately, there are also some problems [Tal, Navathe, Graham [22]] :

1. It needs a method to control and support a large amount of inter-processor communication.
2. The handling of arrays, data structures, and large static databases is often inefficient, especially in a ring architectures.
3. The law of granularity further impacts an already crowded communication bandwidth; granularity is defined as the size of a piece of code obtained by partitioning a problem.
4. The absence of explicit storage imposes serious overhead problems such as the need to circulate stored constants and literals.
5. Operand accumulation causes storage and retrieval congestion.
6. The data-driven philosophy prevents lookahead and instruction overlap parallelism.
7. A process's data flow graph representation is limited to the size of the physical program storage.

Data flow machines are still in the prototyping stage. For example, at M. I. T. a group is working on the Monsoon Project. In Japan, a group is developing the Sigma-1 computer.

2. 4. 3 Parallel Random Access Machine

Parallel random access machine (PRAM) is a theoretical model. In this model, we have a shared, global memory, which all processors can read from or write to "in parallel". Of cause each processor can also perform various arithmetic and logical operations in parallel.

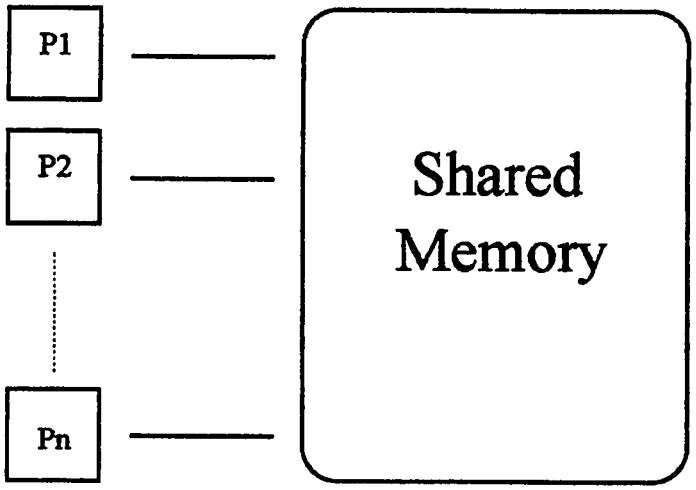


Figure 2.15: A parallel random access machine (PRAM).

Figure 2.15 is a basic architecture of the PRAM. There are ***P*** processors connected to a shared memory. Each processor can access an arbitrary word of the shared memory at any time with unit cost. Obviously, the PRAM model is somewhat unrealistic. The assumption that any processor can access any word of the memory at the same time will also create serious contention problem. Different models of PRAM are classified based on the way they handle memory conflicts. What would happen if more than one processor tried to read from the same memory location? Or what if more than one processor tried to write to the same memory location at the same time? What should be put into that location? There are four different ways to handle these problems.

EREW - exclusive read and exclusive write

CREW - concurrent read and exclusive write

ERCW - exclusive read and concurrent write

CRCW - concurrent read and concurrent write

Exclusive here means only one processor can read from or write to the same memory location at a time. Usually we won't consider the ERCW model, because concurrent write is considered to be much more difficult to handle. If concurrent write is allowed, it would seem redundant to exclude concurrent read.

In the concurrent write models, the problem of write conflict still exists when more than one processor tries to write to the same location at the same time. The value actually stored at that memory location can be determined by three alternative ways:

1) Arbitrary : an arbitrary value from among those written is actually stored.

2) Priority : value from the processor with the highest priority is stored.

3) Combination : the value being stored is some specified combination of the values being written.

The PRAM is widely used as a introductory and instructional tool for parallel computing because of its simplicity.

Chapter 3. Parallel Sorting

3. 1 Introduction

Sorting is one of the most common tasks performed on a computer. Almost all applications maintain their data structures in some order. For example, a heap is a complete binary tree where the parent nodes are always greater than their descendants. Efficient access to information depends on the maintaining of this order.

Sorting is also the foundation of many important algorithms. For example, the solutions of many optimization problems require the solution spaces to be sorted, e.g. optimal solution for the knapsack problem by greedy method requires the rate of cost to profit to be sorted. The performance of the algorithms relies on the correctness and efficiency of the sorting algorithms. To improve the performance of the algorithms, it is necessary to improve the performance of the sorting routines being incorporated into the algorithms. As a result, sorting has became a major target for the introduction of parallelism.

On the other hand, sorting has additional importance to parallel computing. For example, the write requests have to be sorted when emulating a CRCW PRAM by an EREW PRAM. And sorting is frequently used to perform general data permutation on distributed memory computers. Furthermore, sorting is closely related to the task of routing data among processors.

This chapter will cover several important parallel sorting algorithms. Some of them are parallelized versions of their sequential counterparts. Some of them are designed specifically for parallel machines. Before any discussion and analysis of parallel sorting

algorithms can proceed, we first must examine how to measure the performance of a parallel algorithm and how much improvement can be achieved by a parallel algorithm.

3. 2 Measuring the Performance of a Parallel Algorithm

There are several criteria to measure the performance of a parallel algorithm.

1. We can measure the total running time of the algorithm.
2. We can count the number of processors required in the algorithm.
3. Sometimes it is useful to compare the total running time of the parallel algorithm to the total running time of the best sequential algorithm. This is called the speedup of the parallel algorithm, and is defined to be:

$$Speedup = \frac{Total\ time\ of\ the\ best\ sequential\ algorithm}{Total\ time\ of\ the\ parallel\ algorithm}$$

For example, it has been shown that any sequential sorting algorithm using comparison has time complexity of at least O(N log N). If there is a parallel sorting algorithm which has time complexity of O(log N), then the speed up of the parallel algorithm is O(N log N) / O(log N) = O(N).

Obviously, it is desirable to develop parallel algorithms which have as much speedup as possible since a parallel algorithm requires more resources and overhead. In particular, for a parallel algorithm requiring P processors, we would like the parallel algorithm to be P times faster than the best sequential algorithm. When an algorithm achieves that kind of performance, we say that the parallel algorithm achieves linear speedup.

Note that a parallel algorithm can never achieve better than linear speedup. Otherwise, we can simulate the parallel algorithm with a sequential machine by performing each parallel computation sequentially. Thus, achieving a better sequential running time by creating a contradiction to the claim that the original algorithm is the best sequential algorithm.

For example, the best sequential sorting algorithms have time complexity of O(N log N). With a parallel algorithm using N processors, the best we can hope for is a speedup to O(log N).

4. Another important measure of the performance of a parallel algorithm is its efficiency. The efficiency is measured by the work performed by the algorithm. The work of an algorithm is defined to be:

Work = Total running time * Number of processors

For example, an algorithm with running time O(log N) using N processors. The work of the algorithm is O(log N) * N = O(N log N).

Using the notation of work, efficiency can be measured by the ratio of the running time of the best sequential algorithm to the work of the parallel algorithm. Therefore, the efficiency of the parallel algorithm is:

$$\textit{efficiency} = \frac{\textit{running time of the best sequential algorithm}}{\textit{work of the parallel algorithm}}$$

According to the rule of linear speedup, running time of the best sequential algorithm will always be smaller than the work of the parallel algorithm. Therefore, the efficiency of a parallel algorithm can never be bigger than 1.

3. 3 Lower Bounds on Parallel Sorting

To analyze the performance of a parallel algorithm, it is important to realize the limits of the architectures being used. Processor arrays and shuffle-exchange networks are the most practical and commonly used parallel models. The following are some lower bounds for sorting on those models [Quinn [14]].

For the following theorems, we will assume that N elements are to be sorted and the elements are to be distributed evenly before and after the sort, one element per processor.

Theorem 3. 1

For a one-dimensional processor array, the lower bound on the time complexity is $\Theta(N)$.

Theorem 3. 2

For a two-dimensional processor array, the lower bound on the time complexity is $\Theta(\sqrt{N})$.

Theorem 3. 3

Let $N = 2^k$. For a shuffle-exchange network, the lower bound of the time complexity is $\Theta(\log N)$.

Now, we can start to look at some parallel sorting algorithms.

3. 4 Sorting on a Linear Processor Array

We start with a sorting algorithm that works on the simplest parallel architecture, a linear processor array. The algorithm discussed in this section is based on the implementation in [Quinn [14]]. Variations of this algorithm can be found in [Leighton

[11]]. A linear processor array (Figure 3.1) is a linear list of processors; each interior processor is connected to its left neighbor and right neighbor with bi-directional links. The first and last processor in the linear array may have just one connection, and may serve as input/output points for the whole array.

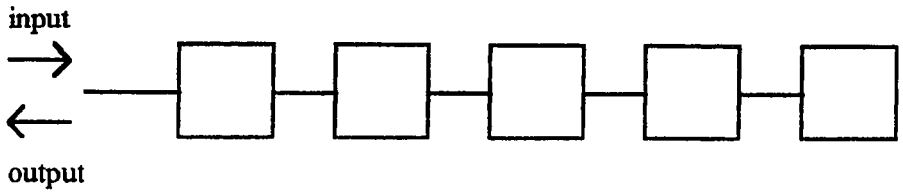


Figure 3.1 : A 5-cells linear processor array.

Each processor consists of a local program control and a local memory. The complexity of the local control and the size of the local memory may vary. In a general array processor, the control program is usually simple (i.e. only a few operations are available) and the size of the local memory is usually small.

The linear array algorithm uses a systolic approach; the entire processor array is synchronized by a global clock. As in all systolic algorithm, data pulses through the processors similar to the way blood pulse though the body. At each step, each processor:

1) obtains input from its neighbor processor;

2) inspects its local memory;

3) performs the computation issued by its local control program;

4) outputs to its neighbor; and

5) updates its local memory.

To sort N numbers, a N-cell linear array will be used. During each step of the algorithm, each processor will perform the following:

1) Receive the input from its left neighbor.
2) Compare the input with the value in its local memory.
3) Output the larger value to its right neighbor.
4) Store the smaller value in its local memory.

After 2N-1 steps, the values will be sorted where the i-th smallest value is stored in the i-th processor's local memory. To output the sorted values, the simplest way will be for each processor to pass the value in its local memory to its left until the N-th value is output. This will take N-1 steps. Figure 3.2 demonstrate this algorithm on an input of (4, 2, 5, 9, 1).

The total running time of this algorithm is 2N-1+N-1 = 3N -2 = O(N). Comparing with the fastest sequential algorithm, O(N log N), this algorithm has a speedup of O(N log N) / O(N) = O(log N).

However, this algorithm requires N as the number of processors. Therefore, total work of the algorithm is $O(N^2)$. As a result, this algorithm is not work efficient. The reason is that some of the processors are left idle in each step. Therefore, a lot of resources have been wasted.

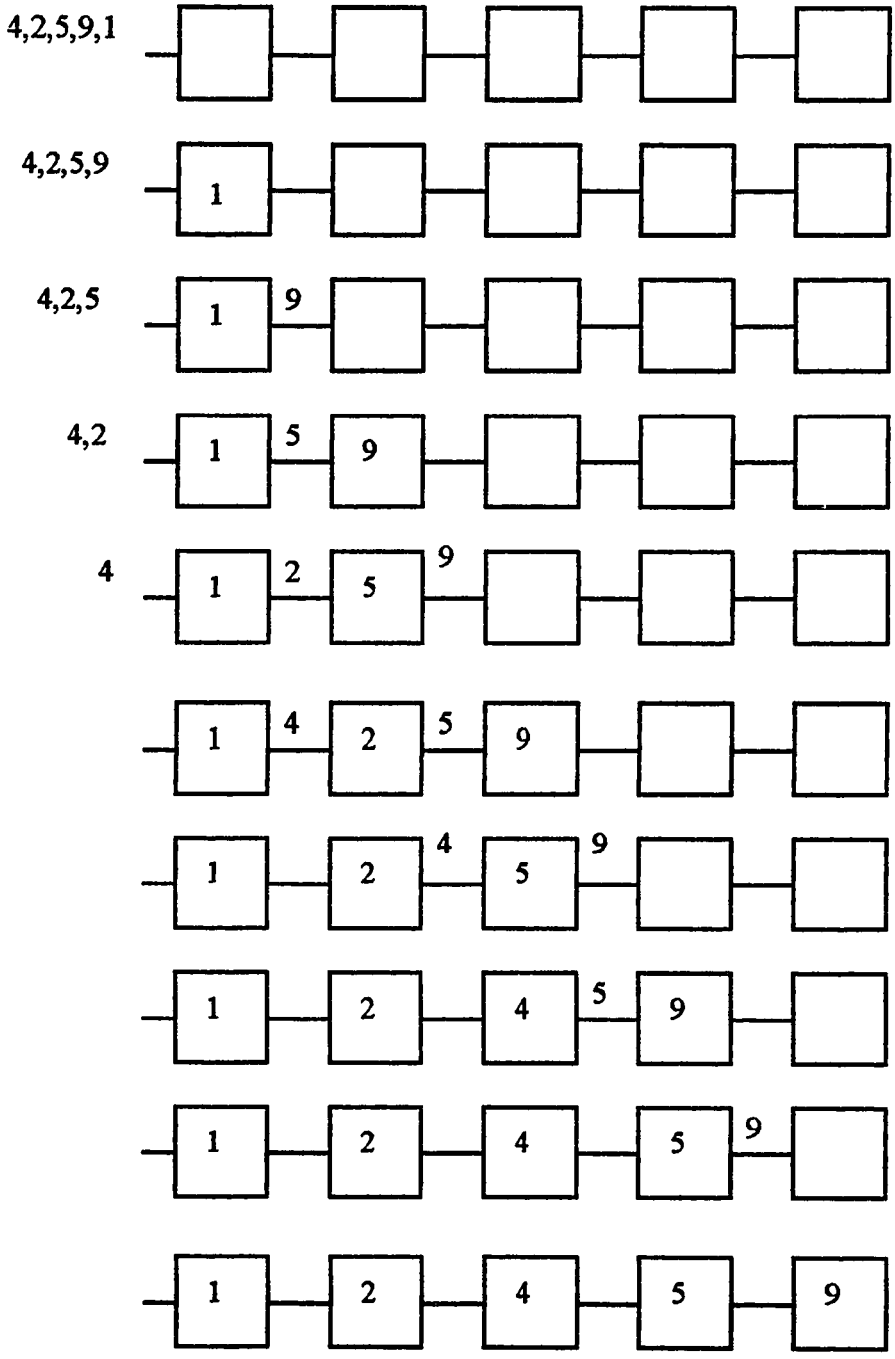


Figure 3.2: Linear array sorting with input (4,2,5,9,1).

3. 5 Enumeration Sort

Given a list of N elements, (a_1, a_2, a_3, ... , a_N). Assume the elements are to be sorted in ascending order, an enumeration sort computes the final position in the sorted list for each element a_i by counting the number of elements in the list that have a smaller key value than a_i. References to this algorithm can be found in [Quinn [14]] and [Akl [2]].

Without the lost of generality, we will assume all the elements are distinct. That is, if there are four elements smaller than a_i, then a_i must be the fifth smallest element.

Since each element has to be compared with N-1 elements, this algorithm requires (N-1)*N comparisons. It is not difficult to see that each comparison is independent. Therefore, all the comparisons can be performed simultaneously. To implement that on a CRCW PRAM, each comparison will be handled by a distinct processor. A counter has to be maintained for each element. Assuming processor i is assigned to compute if $a_j < a_i$, processor i will perform the comparison, increment the counter for a_i if $a_j < a_i$.

The CRCW PRAM is chosen because simultaneous read and write to the same location is allowed in this model. As a result, this algorithm takes constant time.

Algorithm 3. 1 is the enumeration sort where key values of the elements are assumed to be distinct. If the elements can have the same value, the algorithm can be modified such that each processor will be assigned an index. If $a_i = a_j$, and the index of $P_j < P_i$, then the counter will be incremented.

Algorithm 3.1 Enumeration sort

```
Enumeration_sort( )
int n; /* Number of element. */
int a[0. . n-1]; /* Elements to be sorted. */
int counter[0. . n-1]; /* counter[i] = number of elements that is smaller than a_i. */
int sorted[0. . n-1]; /* Sorted elements. */
{  for (i = 0; i < n; i++)
     for (j = 0; j < n; j++)
       for all processing elements PE_i,j do in parallel{
         counter[i] = 0
         if a[i] < a[j]
           counter[i] = counter[i] + 1
   }
   for (i = 0; i < n; i++)
     for all processing element PE_i,0 do in parallel
       sorted[ counter[i] ] = a[i]
} /* End procedure Enumeration_sort. */
```

This algorithm achieves O(1) running time because it uses a powerful and some what unrealistic model. Still, the efficiency of the enumeration sort is not that impressive since it uses $O(N^2)$ processors. Thus, the work done by this algorithm is $O(N^2)$. Comparing to the best sequential algorithm, the efficiency of the enumeration sort is $O(N/\log N)$.

However, all the comparisons are independent, and there can be a trade-off between the amount of resource and the running time. We can reduce the number of processors by sacrificing running time. In fact,

Running time = N^2 / Number of processors

Therefore, there is a certain flexibility in this algorithm. And the algorithm can accept any size input.

3. 6 Odd-Even Transposition Sort

The odd-even transposition sort is designed for a one-dimensional mesh processor array. More references of this algorithm can be found in [Quinn [14]] and [Leighton [11]]. In the processor array, each processor can access two values, a unique element of the unsorted list and a value retrieved from its neighbor.

To sort N elements, the sorting algorithm takes N/2 iterations, each iteration consists of two phases: odd-even exchange phase and even-odd exchange phase. In the odd-even exchange phrase, odd-numbered processors compare their assigned values with that of their next higher even-numbered processors. If they are out of sequence, these two values are swapped. In the even-odd exchange phrase, even-numbered processors carry out similar operations with their odd-numbered counter-parts. Assuming that the values are to be sorted in ascending order, the lower-numbered processors will contain the smaller values. After N/2 iterations (N steps), the values will be sorted. The algorithm is presented as Algorithm 3.2 and Figure 3.3 is an example of the sorting operations with input (8,3,7,4,2,9,1,5). Note that this algorithm corresponds to the sequential bubble sort algorithm.

Algorithm 3. 2 Odd-even transposition sort

```
Odd-even_transposition_sort( )
 int a[0. . n-1]; /* Elements to be sorted.  */
{
  for i = 1 to n/2 do
    for (j = 0; j < n-1; j++)
     for all processing elements PE_j do in parallel {
       if  ( (j < n - 1) and odd(j) )
         if a[j] < a[j+1]
           swap( a[j], a[j+1] )
       if even(j) then
         if a[j] < a[j+1]
           swap( a[j], a[j+1] )
  }
} /* End procedure Odd-even_transposition_sort.  */
```

The running time of the odd-even transposition sort is O(N) using N processors.
The total work is N * O(N) = O(N^2).
The efficiency = O(N log N) / O(N^2) = O(log N / N).

The reason for the inefficiency is that, for each step, half of the processors are idle. Another limitation is that it moves elements only one position at a time. If the elements are close to being sorted, and the out of order elements are O(N) from their proper positions, this algorithm still takes O(N^2) work. Clearly, it is not work efficient. However, since

the lower bound for sorting N elements on a one-dimension mesh is $\Theta(N)$, odd-even transposition sort is optimal for this model.

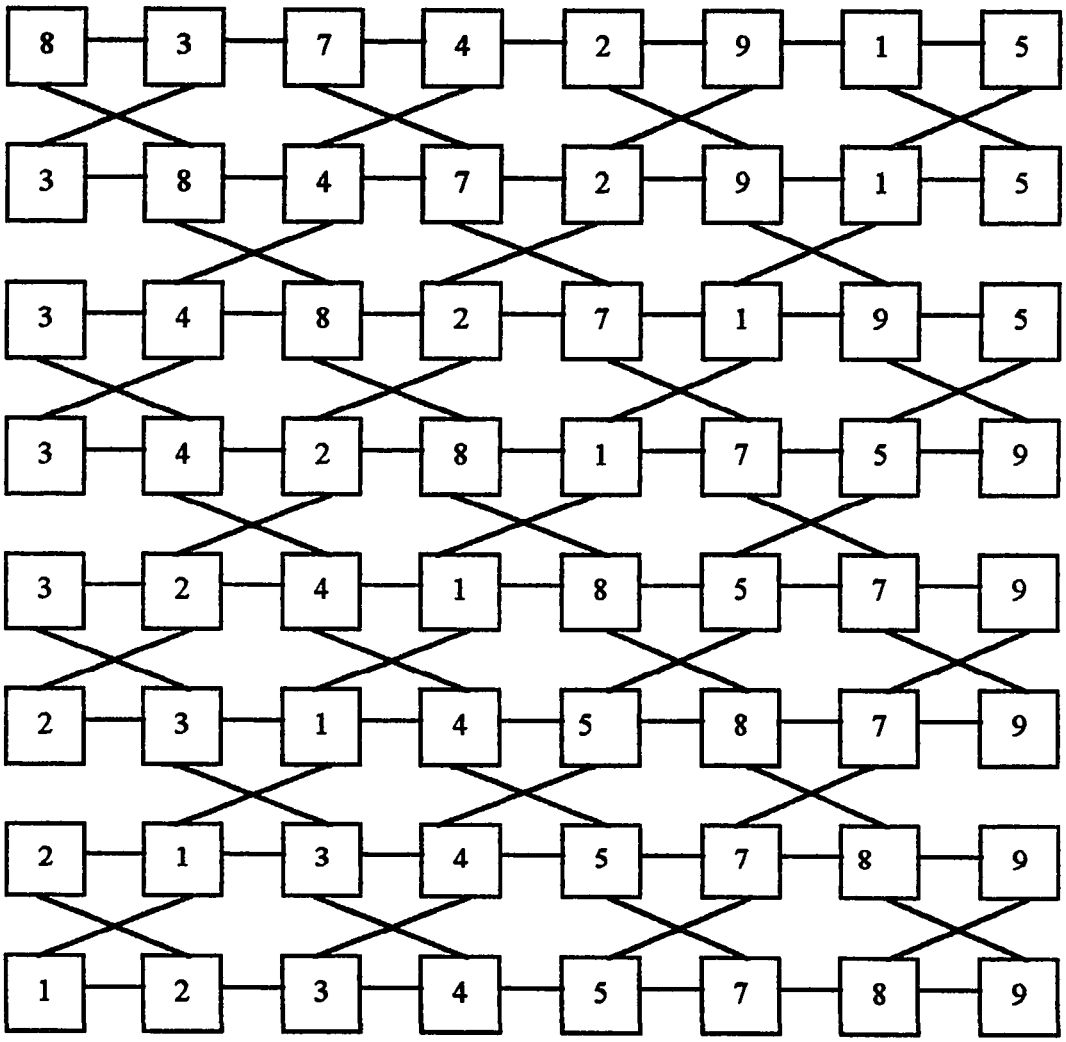
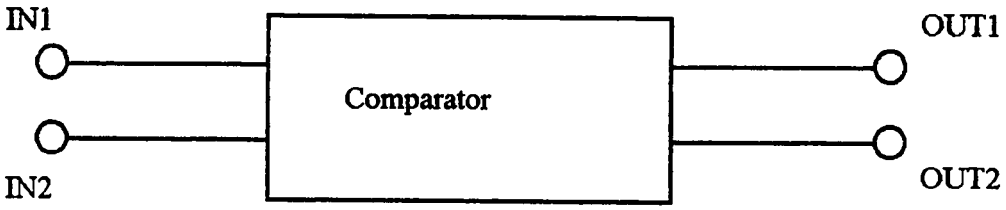


Figure 3.3: Odd-even sorting with input (8,3,7,4,2,9,1,5).

3. 7 Batcher's Odd-Even Sorting Network

The odd-even sorting network is based on the idea of the sequential merge sort. It uses a network of comparators. The merging network can be constructed recursively as in this section [Schmeichel [17]] or it can be implemented with a butterfly network [Leighton [11]].

First, let's define the most basic level of the network - the comparator (Figure 3. 4). Each comparator accepts two inputs. It returns the larger input as output1 and the smaller input as output2. In other words, it sorts two inputs.



OUT1 = min(IN1, IN2)

OUT2 = max(IN1, IN2)

Figure 3.4: A comparator

Next, we need a merging network to merge two equal size sorted sequences into a single sorted sequence. The merging network also can be defined recursively (Figure 3.5a, b, c, d).

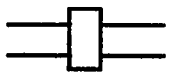


Figure 3.5a: Size 2 merging network

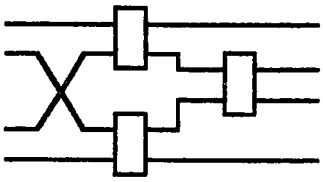


Figure 3.5b: Size 4 merging network.

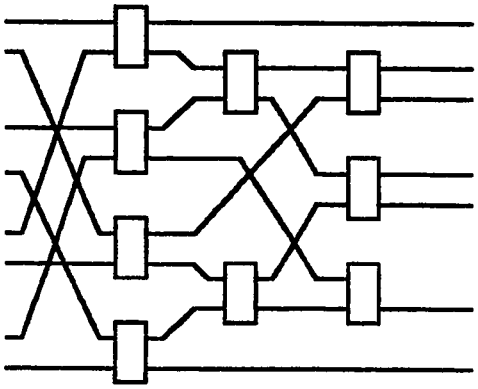


Figure 3.5c: Size 8 merging network.

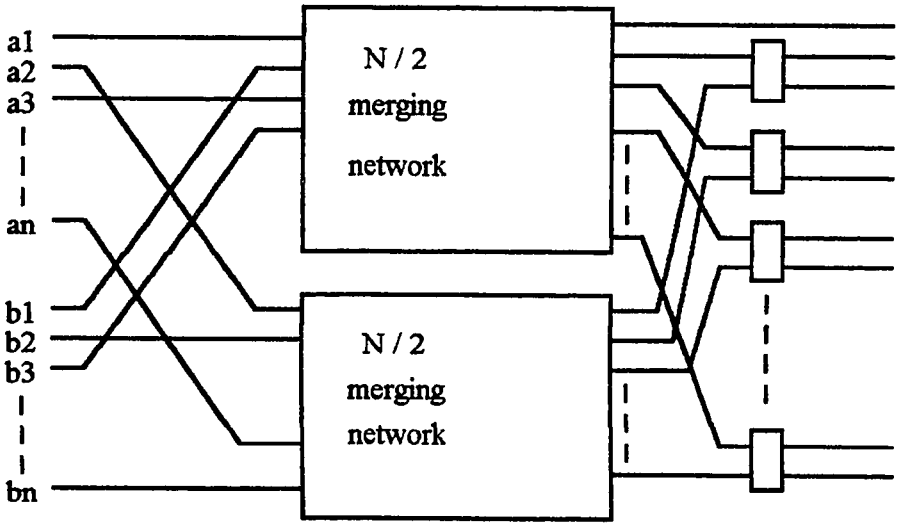


Figure 3.5d: Size n merging network.

Each merging network consists of networks of smaller size, therefore, the correctness of the network can be proved by induction. It is obvious that the size 2 merging network works. Using induction, we can prove the size n merging network works.

The number of processors used in this merging algorithm corresponds to the number of comparators used. Let the number of comparators used be $C_m(N)$,

$C_m(N) = 2C_m(N / 2) + (N-1) = O(N \log N)$ [Schmeichel [17]].

The speed of the comparator is technology dependent. To measure the speed of the comparator network, we will measure the depth of the network which is proportional to the speed of the network. Let the running time of the merging network be $T_m(N)$,

$T_m(N) = T_m(N / 2) + 1 = O(\log N)$ [Schmeichel [17]].

With the above merging network, we can build the sorting network recursively (Figure 3.6a, b, c). Since the merging network works, the sorting network in turn works correctly. We can use an approach similar to the analysis of the merging network to measure the complexity of the sorting network.

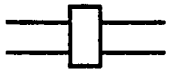


Figure 3.6a: Size 2 sorting network.

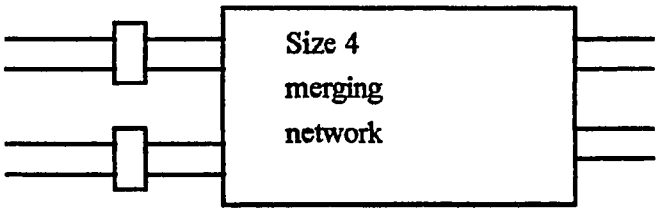


Figure 3.6b: Size 4 sorting network.

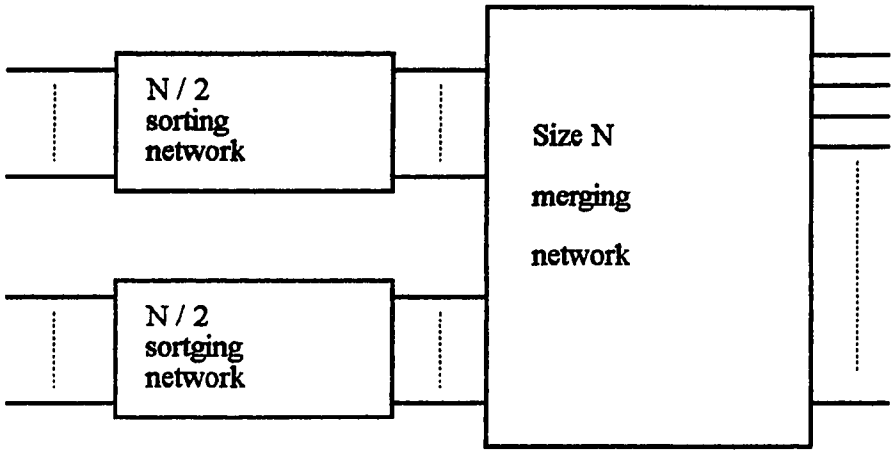


Figure 3.6c: Size N sorting network.

Let the number of comparators used be $C_S(N)$,

$C_S(N) = 2C_S(N / 2) + O(N \log N) = O(N \log^2 N)$ [Schmeichel [17]].

Let the running time be Ts(N),

$T_S(N) = T_S(N / 2) + O(\log N) = O(\log^2 N)$ [Schmeichel [17]].

Total work $= O(N \log^4 N)$ and efficiency $= 1 / O(\log^3 N)$.

It is not work efficient because only one level of comparators are active at a time. As a result, many comparators are left idle. Since any sorting algorithm using comparison take at least O(N log N) comparisons, the sorting network needs at least O(N log N) comparators. One way to improve the running time of the sorting network is to reduce the depth of the network; however, this is not easy to achieve. If we simulate the sorting network on a multi-processors system, each comparator can be simulated by a processor. There is no reason we cannot use the same processors to simulate different level of comparators. Therefore, a lot of resources can be saved. Of course, additional logic has to be added to the system to reuse the processors at different levels.

3. 8 Batcher's Bitonic Merge Sort

Another parallel sorting algorithm, introduced by Batcher, is the Bitonic merge sort algorithm. A more in depth discussion of the algorithm can be found in [Smith [21]]. The bitonic merge sort utilizes the properties of a bitonic sequence. A bitonic sequence is a sequence of values satisfying either one of the following properties:

1) It starts out monotonically increasing up to some point, then monotonically decreasing.

2) It starts out monotonically decreasing up to some point, then monotonically increasing.

For example, {1, 3, 5, 8, 9, 7, 6, 4, 2} is bitonic , where {1, 3, 5, 4, 2, 6, 8, 7, 9} is not. If the sequence is represented graphically, it means the graph can only have one peak or one valley (Figure 3. 7).

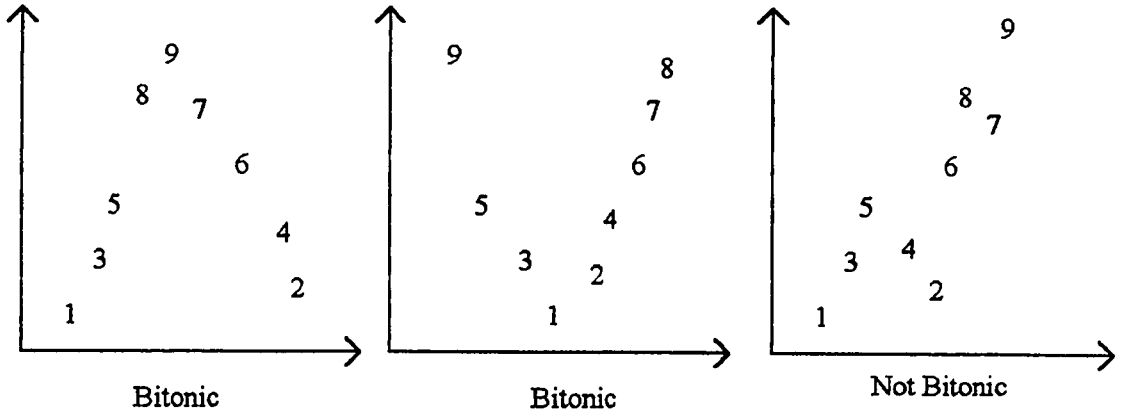


Figure 3.7: Seqences of number represented graphically

Given a bitonic sequence { b_0, b_1, b_2, ... , b_{n-1} } of size N, where N = 2m. A bitonic halver do the following compare-exchange operation:

for i = 0 to m do in parallel

if ($a_i < a_{i+m}$) then swap(a_i, a_{i+m})

Notice that the bitonic halver performs some limited sorting of its input. After the input $\{b_0, b_1, b_2, \ldots, b_{n-1}\}$ goes through the bitonic halver to produce the output $\{d_0, d_1, d_2, \ldots, d_{m-1}, e_0, e_1, e_2, \ldots, e_{m-1}\}$, the sequences $\{d_0, d_1, d_2, \ldots, d_{m-1}\}$ and $\{e_0, e_1, e_2, \ldots, e_{m-1}\}$ is either sorted or bitonic. The bitonic sorting algorithm can be constructed using the bitonic halver:

Let $\{b_0, b_1, b_2, \ldots\ldots, b_{n-1}\}$ be a bitonic sequence, $n = 2^k$
for $i = k-1$ downto 1 do in parallel
begin
divide the sequence into sublists of size 2^i
perform the bitonic halving on each sublist
end

For example, to sort { 1, 3, 5, 8, 9, 4, 3, 1 }:
after 1st bitonic halving : { 1, 3, 3, 1, 9, 4, 5, 8 }
after 2nd bitonic halving : { 1, 1, 3, 3, 5, 4, 9, 8 }
after 3rd bitonic halving : { 1, 1, 3, 3, 4, 5, 8, 9 }
Since each bitonic halving can be carried out in a single step, the running time of bitonic sort is O(log N) using O(N) processors.

However, the bitonic sort only works on a bitonic sequence of inputs and the input for a sorting application is rarely bitonic. The important point of bitonic sort is it gives rise to a very efficient algorithm to merge two sorted lists. All it takes is to reverse one of the sorted lists and concatenate it to the end of the other list, then we have a bitonic sequence. At this time, the bitonic sort can be applied. We will called this the Batcher's merge algorithm.

Using the Batcher's merge algorithm, we can derive a very efficient sorting algorithm. We will call it the Batcher's bitonic merge sort. The Batcher's sort consists of two phases:

1. Sort the left and right halves of the sequence (recursively in parallel).
2. Do a Batcher's merge of the two sorted sublists.

After simple analysis of the algorithm, we have:

Running time of the Batcher's sort = $O(\log^2 N)$.

Number of processors used = $O(N)$.

Total Work = $O(N \log^2 N)$.

Efficiency = $O(1 / \log N)$.

Since the optimal sequential sort using comparison requires $O(N \log N)$ running time, the Batcher's sort is very close to optimal. Another point that needs to be stated is that the bitonic sort can be implemented on a network of comparators.

3.9 AKS Sorting Network

AKS stands for the names of the three Hungarian scientists - Ajtai, Komlos, Szemeredi - who developed this sorting algorithm. [Ajtai, Komlos, Szemeredi [1]]

It is the first optimal sorting network that achieve O(log N) depth. Nevertheless, it isn't an O(log N)-time sorting algorithm in the sense of the Cole sorting algorithm [Cole [6]]. Given a value of N, we can construct a sorting network with O(log N) depth, and this network varies for different values of N. In other words, we have a different algorithm for each value of N.

There are many different versions of the three Hungarian's algorithm. The idea in this section is based on the interpretation of [Smith [21]]. Another good interpretation is the Paterson's version.

The AKS sorting network is able to achieve optimal time because it utilizes the properties of a special kind of graph called the expander graph. Before discussing the AKS network, we will define the expander graph.

Definition 3. 9. 1

Given a graph G = (V, E), and a set S of vertices. $\Gamma(S)$ is defined to be the set of neighbors of S. Let G be a bipartite graph, the two sets of nodes of the bipartition are V_1, V_2, and $|V_1| = |V_2| = n$. G is an (k, β_1, β_2) expander graph if:

(i) $\forall x \in V_1$ or V_2, degree of $x \leq k$, and

(II) $\forall X \subset V_1$ or V_2, $|X| \leq n\,\beta_1$ and $|\Gamma(X)| \geq \beta_2 |X|$.

In other words, every sufficiently small ($|X| \leq n\,\beta_1$) set of nodes has "many" ($|\Gamma(X)| \geq \beta_2 |X|$) neighbors. Figure 3.8 is a (3, 3, 1/4) expander graph.

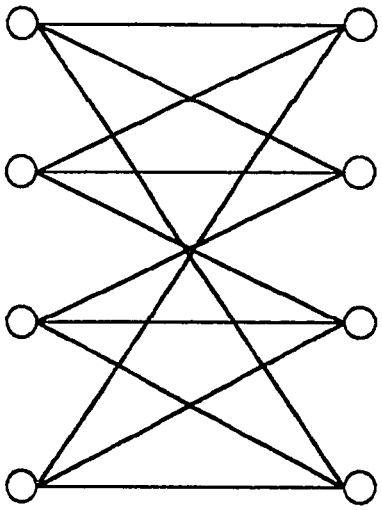


Figure 3.8: An (3, 3, 1/4) expander graph.

Next, we need to define what the 1-factor of a graph is.

Definition 3. 9. 2

Let G = (V, E), a 1-factor of G is a set S = { e_1, e_2, ... , e_k } of disjoint edges that span the graph. Figure 3. 9 is the three 1-factors of the expander graph in Figure 3.8.

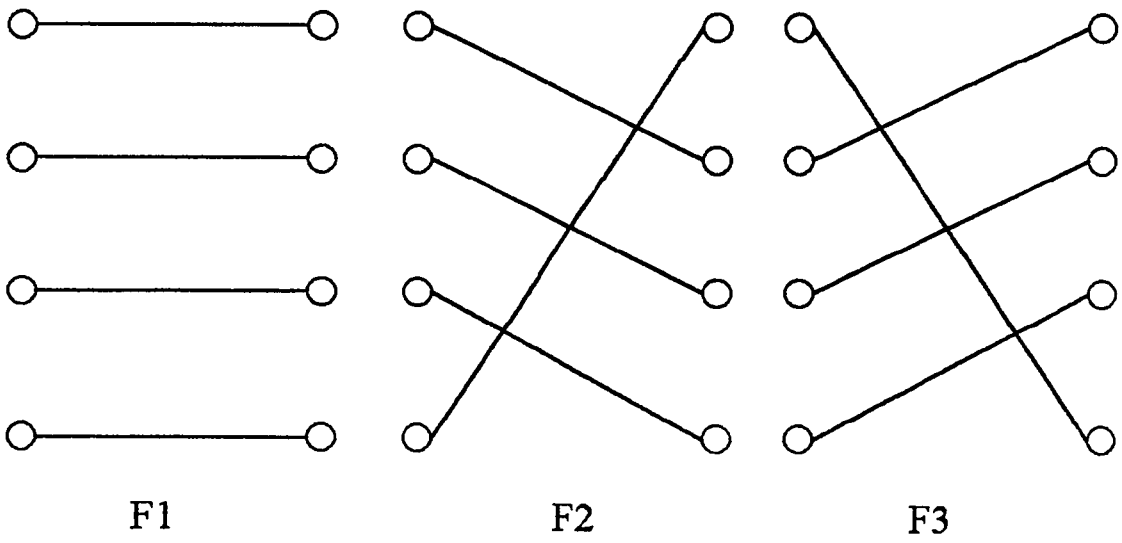


Figure 3.9: The partition of G in Fig. 3.8 into three 1-factors, F1, F2 and F3.

With the above definitions, we can start describing the AKS algorithm. The AKS sorting algorithm requires the use of two sub-routines, ε'-halving and ε-nearsort. We will first describe ε'-halving.

ε'-halving

Given an input of N numbers, first construct an expander graph G with N nodes. Next, assign the top half and the bottom half of the input to each of the bipartitions. Then,

decompose G into k different 1-factors : { F_1, F_2, ... , F_k }. After the 1-factors are constructed, carry out the following iterations:

For i:= 1 to k do

For all edges e ∈ F_i, do in parallel

compare numbers at the ends of e, interchange them if they are out of order

This requires at most k parallel operations. Using the expander graph in Figure 3.8 with input (6,5,7,3,4,8,1,2), we will carry out the ε'-halving operation in Figure 3.10. The idea is after an ε'-halving, all the values in one side of the bipartitions will be larger than the values in the other side of bipartitions with some degree of error.

To be more specific, an ε'-halving is "correct" if and only if each element in one bipartition is no larger than any element in the other bipartition. Any element wrongly placed by the ε'-halving is said to be an "error". For the example in Figure 3.10, there is an error in each bipartition, namely elements 5 and 4. However, we will see ε'-halving is always, within a known tolerance, approximately correct. In fact, we have the following theorem:

Theorem 3. 9. 1 [Smith [21]]

Let g be an (k, ε', (1-ε')/ε') expander graph. After performing an ε'-halving using G, the number of errors in each bipartition of G is at most ε'n/2. Therefore, the total number of errors is at most ε'n.

It is also true that the ε'-halving operation works with reasonably good accuracy at the ends of the input sequence. Most of the errors are concentrated in the middle. Using the ε'-halving, we can develop an approximate sorting algorithm called ε-nearsort.

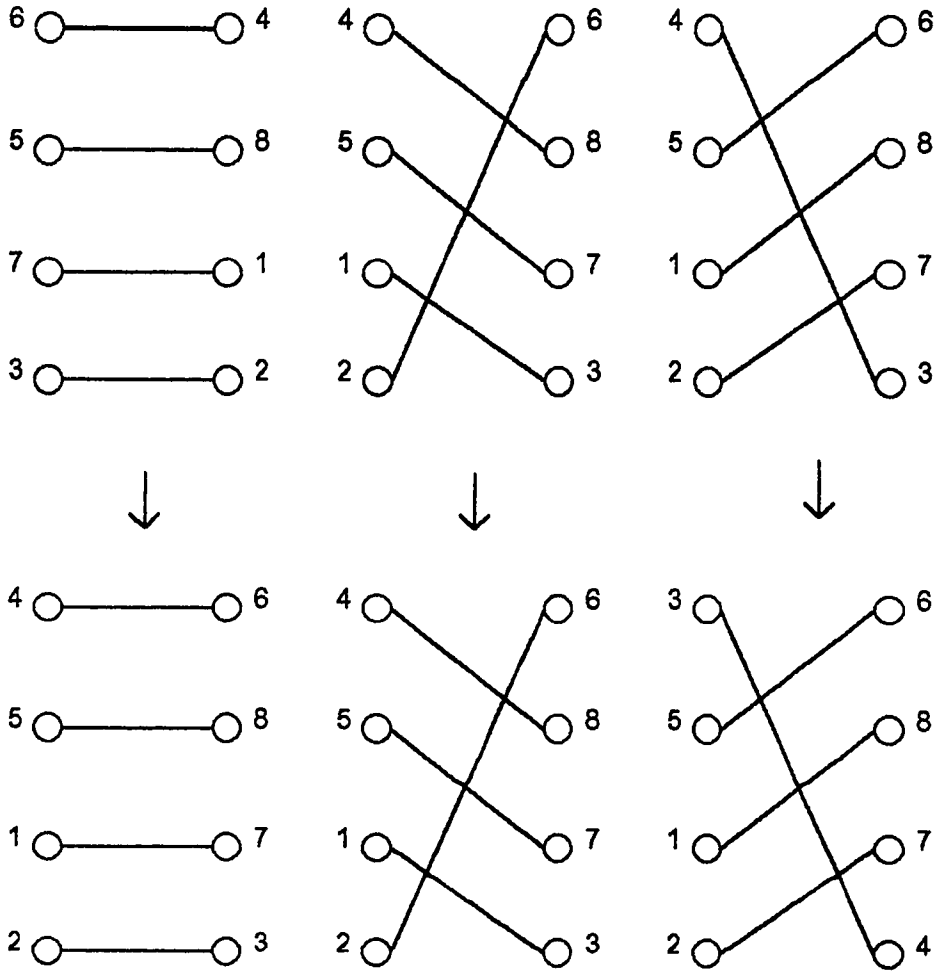


Figure 3. 10 ε'-halving of input (6,5,7,3,4,8,1,2) using the expander graph in Figure 3. 8

ε-nearsort

Given a set of n numbers, first apply an ε'-halving to the whole set of numbers. Next, apply an ε'-halving to the top and bottom of the result. Then, apply the ε'-halving to each quarter, eighth, etc. , until each piece has size < nε'.

This executes in constant time because the algorithm terminates when the pieces become a fixed fraction of the size of the original set of numbers. For each piece of size w = nε, there is at most εw errors.

Now, we have an approximate sorting algorithm that runs in constant time with guaranteed error rate. One intuitive idea is if we can reduce the number of errors by a constant factor of ε each time we perform an ε-nearsort, we may be able to get an exact sorting algorithm by performing the ε-nearsort repeatedly and preferably in O(log N) time.

Simply applying the ε-nearsort repeatedly may not work since it may make the same errors each time we run the algorithm. The algorithm developed by Ajtai, Komlos and Szemeredi applies the ε-nearsort repeatedly in such a way that the errors get corrected. And it turn out that this requires the ε-nearsort to be performed O(log N) times. With all the fundamentals stated, now we will describe the AKS sorting network.

AKS Sorting Algorithm

Let the inputs be stored in some memory locations. The algorithm is divided into log N iterations. Each of the iteration consists of three steps. In each of these steps, we perform an ε-nearsort separately and simultaneously on each set of a partition of the memory locations.

The way which these memory locations are partitioned will be easier to understand if it is represented as a binary tree. Consider a complete binary tree of log N depth, each vertex of the binary tree corresponds to a natural interval of memory locations. The natural interval of memory locations is defined to be an equal subdivision of the N memory locations by the vertices at each level of the binary tree. For example, the natural interval of the root node will be all of the memory locations; the natural interval of the vertices at level 1 (assuming the root is at level 0) of the tree will be an equal subdivision of the N memory locations, i.e. memory locations from 1 to N/2 and locations from N/2 + 1 to N. Figure 3.11 is a complete binary tree of depth 3 and each vertices' natural intervals.

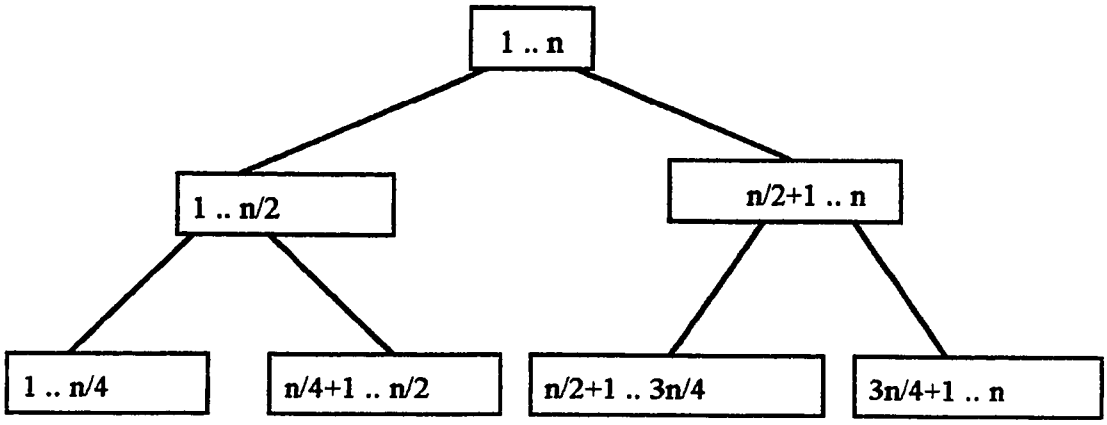


Figure 3.11: A complete binary tree of depth 3, with natural intervals.

In addition to the definition of the natural interval, we also need to include a parameter A which satisfies the condition that $A << 1/\varepsilon$ (<< denote sufficiently smaller than). In general, A must also satisfies $\varepsilon < A^{-4}$. Now we can describe how the memory locations are distributed among the vertices of the binary tree in each phase *i* of the algorithm. The distribution of memory locations can be divided into two steps:

1. Initial assignment step:

- Vertices of depth $j > i$ have an empty set of memory locations assigned to them.
- Vertices of depth i have their natural interval of memory locations assigned to them.
- Vertices of depth $j < i$ have a subset of their natural intervals $\{I_1, \ldots I_k\}$ - from term 1 to $\lfloor A^{-(i-j)} \rfloor * k$ and from term $(k - \lfloor A^{-(i-j)} \rfloor * k)$ to k - assigned to them.

2. Sifting step:

-Each memory location only remains being assigned to the vertices with the lowest depth.

For example, given memory location {1 ... 8 } and parameter A = 4. At the first iteration, depth of the root node = 0, and i = 0, only the root node is being considered. Therefore, it has the whole set of memory locations assigned to it (Figure 3. 12).

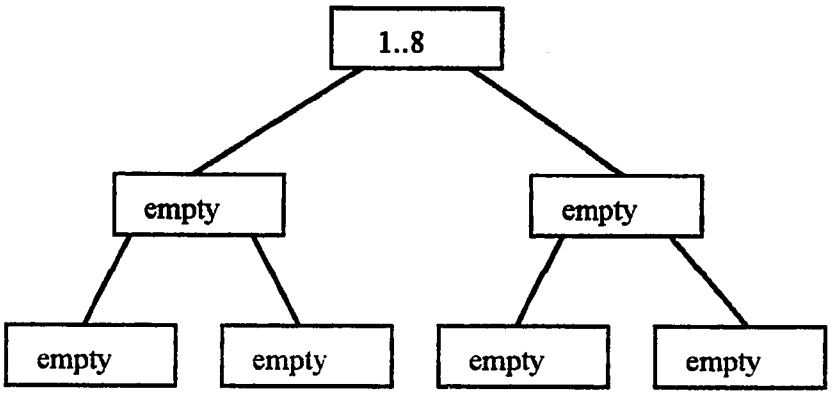


Figure 3.12: Memory locations tree after the 1st iteration.

At the second iteration, the root node has memory locations 1, 2, 7, 8 being assigned to it after the initial assignment step. Its child nodes have the memory locations of their natural intervals assigned to them (Figure 3.13a). After the sifting step, since the root node has higher priority, it got to keep its memory locations while its child nodes got what was left (Figure 3. 13b).

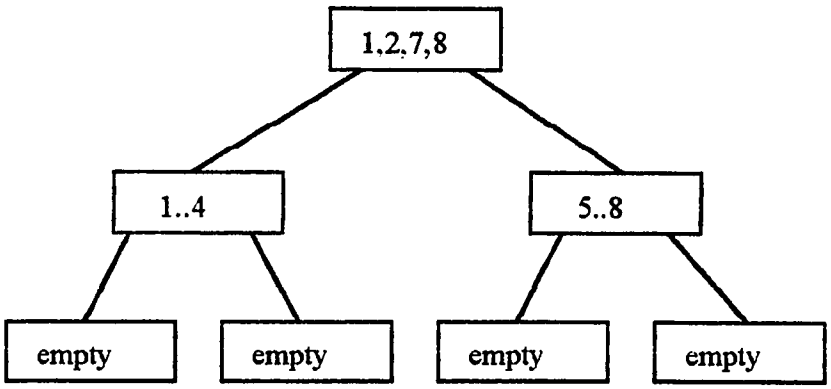


Figure 3.13a: Memory locations tree after initial assignment step of 2nd iteration.

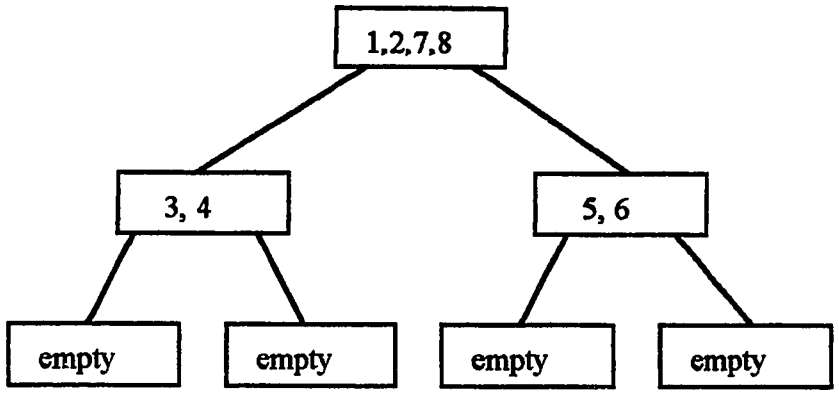


Figure 3.13b: Memory locations tree after sifting step of 2nd iteration..

At the third iteration, the assignment of memory locations filters down to the leaves of the tree. Finally, we get the results of Figure 3.15a and Figure 3.15b.

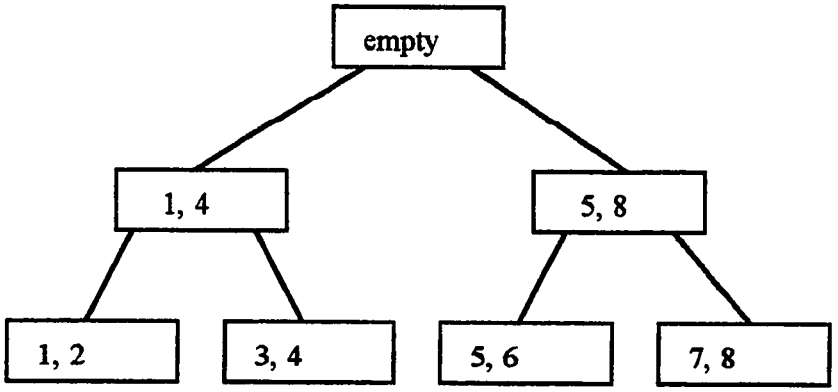


Figure 3.14a: Memory locations tree after initial assignement step of the 3rd iteration.

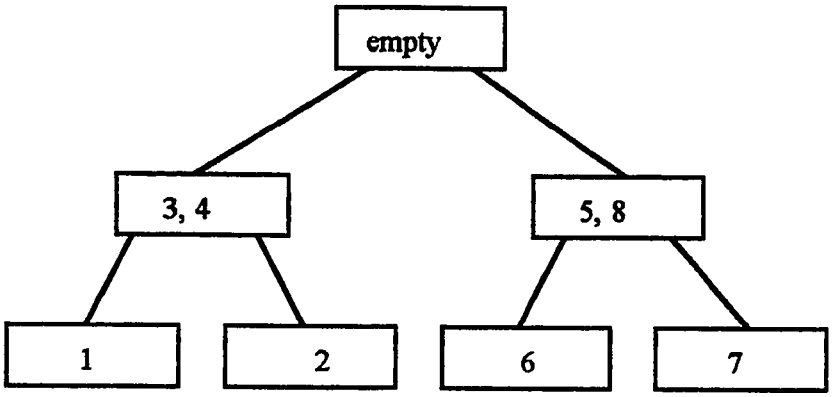


Figure 3.14b: Memory locations tree after sifting step of the 3rd iteration.

After the memory locations are distributed, we will partition them into different sets and apply ε-nearsort operations to them. The way we partition them and apply the ε-nearsort to them can be described in two steps.

Zig-step

Partition the tree into triangles with the apexes at even levels, and perform independent ε-nearsort operations on the sets of memory locations associated with each triangles (Figure 3.15).

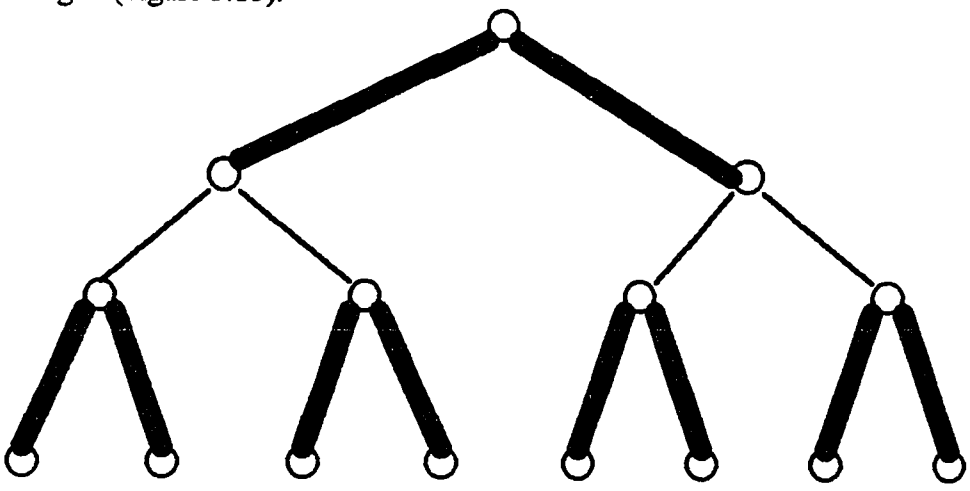


Figure 3.15: Partitions in *Zig* step

Zag-step

Partition the tree into triangles with the apexes at odd levels, and perform independent ε-nearsort operations on the sets of memory locations associated with each triangles (Figure 3.16).

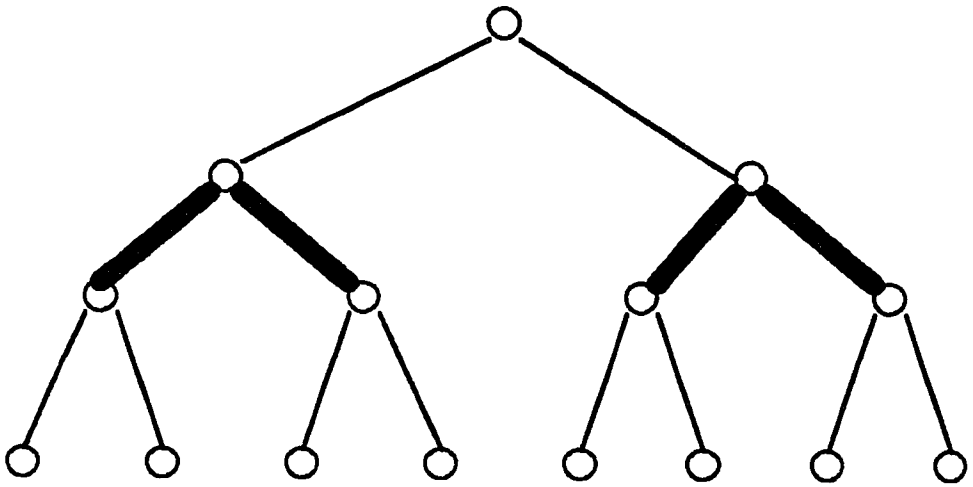


Figure 3.16: Partitions in Zag step.

Finally, we will present the complete AKS sorting algorithm:

AKS sorting algorithm

for (i = 1; i ≤ log N; i++) {

compute the association of memory locations with vertices for phrase i

Zig; Zag; Zig;

}

At the end of this procedure, the input stored in the memory locations { 1 ... n } will be sorted.

After simple analysis of the algorithm [Smith [21]], we have the following results:

Number of processor = O(N).

Total running time = O(log N) since it takes log N iterations and each iteration takes constant time.

Total work = O(N log N).

Efficiency = 1.

This is a sorting network because each ε'-halving can be implemented as a network of comparators, and each ε-nearsort can be implemented as a network using the ε'-halving network.

This algorithm is not uniform. The complexity-parameter N cannot be regarded as one of the inputs to the algorithm. In other words, we don't have an O(log N) algorithm to sort n numbers. We have an algorithm that constructs different O(log N) algorithms for different values of N.

If we don't consider the cost of all the preliminary computations - constructing the expander graphs, partitioning and assigning the memory locations, etc. - this algorithm takes O(log N) time. However, the cost of the preliminary computations can be significant.

The constant factor of the algorithm's time complexity depends upon the knowledge of how to construct the expander graph. The only known explicit constructions of these expander graphs generally lead to very large graphs (i.e. $\sim 2^{100}$ vertices). This means a very large constant factor in the algorithm.

Therefore, the AKS sorting network is generally regarded as essentially a theoretical result - it proves that the optimal complexity is achievable. For problem of practical size, other sorting network such as Batcher's network is faster.

However, there are many known "small" expander graphs. Therefore, we know there should be a better way to construct these expander graphs, which will reduce the size of the graph and in turn reduce the constant factor of the algorithm.

Nevertheless, the AKS sorting network is still a milestone for parallel sorting algorithms. It is the first parallel algorithm to achieve the optimal O(log N) time with N processors. Once this is achieved, we only need to refine the implementation to reduce the constant factor and close the gap between its theoretical importance and its practical use.

Chapter 4. Permutations

4. 1 Introduction

The study of generating permutations has a long and distinguished history. It was one of the first non-trivial non-numeric problems to be attacked using a computer. Back in 1956, Tompkins wrote a paper [Tompkins [23]], describing the use of permutations to solve problems in a number of practical areas. Even though most of the problems described are solved by more sophisticate methods today, his paper stimulated interest in generating permutations by computers.

To formally define the problem of generating permutations, let S be a set of N distinct objects, $S = \{I_1, I_2, I_3, \ldots\ldots, I_N\}$. A m-permutation of S is a list of m objects chosen from S, $m \leq N$, arranged in some order. For example, $\{I_1\ I_2\ I_3 \ldots\ I_m\}$ is a m-permutation. Order of the objects in a permutation is important. Two permutations are distinct if they are differ by the objects in the list or by the orders of the objects in the list. Therefore, e.g. $\{ I_1, I_2, I_3 \} \neq \{ I_1, I_3, I_2 \} \neq \{ I_2, I_4, I_6 \}$.

The number of distinct m-permutations of N objects is denoted by ${}^{N}P_{m}$, and ${}^{N}P_{m} = N! / (N-m)!$. For example, if N = 5, there are 120 distinct 4-permutations.

If m = N, there are N! of permutations. If N = 10, there are 3628800 permutations. This is a very fast growing function. The inherent nature of the problem makes it impossible to generate permutations by hand for even a modest number of objects.

Computers help speed up the time required to generating permutations, but not by as much as one might think (Figure 4. 1). Assuming computer can generate one permutation per μsec, for N = 17, there are 355,689,428,096,000 permutations, and it will

take 10 years for the computer to generate all N-permutations. For N > 25, the time required will be far greater than the age of the earth.

	N!	Time
1	1	
2	2	
3	6	
4	24	
5	120	
6	720	
7	5040	
8	40320	
9	362880	
10	3628800	3 seconds
11	39916800	40 seconds
12	479001600	8 minutes
13	6227020800	2 hours
14	87178291200	1 day
15	1307674368000	2 weeks
16	20922789888000	8 months
17	355689428096000	10 years

Figure 4.1 Approximate time needed to generate all permutations of N. (1 μsec per permutation)

Fortunately, we usually don't want to generate every single one of the m-permutations. Applications are usually only interested in processing some particular subsets of the m-permutations. Generating all the m-permutations before picking out the ones that are wanted will be impractical. If we can generate the permutations in some pre-defined order, it certainly will be easier for the applications to chose the permutations they wanted. In this thesis, we are interested in generating the permutations in lexicographic order because it is the most natural way to enumerate permutations.

Since the time it takes to generate all m-permutations is inherently exponential, the more important measure of performance is the delay time between successive permutations. In this thesis, we will design a parallel algorithm that generates permutations in lexicographic order with efficient delay time and simple architecture. By using parallel architectures, we can execute more than one instruction per clock cycle. However, we also need to revise our algorithms to fully utilize this capability.

Generating permutations in lexicographic order is only a special case of generating permutations. Therefore, it will be useful to look at some sequential and parallel algorithms that generate permutations.

4. 2 Sequential Permutation Algorithms

4. 2. 1 Methods Based on Exchanges

A natural way to generate a new permutation from an existing permutation is to exchange two elements of the old permutation. First, we will assume that there exists an operation which exchanges two elements - exchange(element_1, element_2). For example, exchange(P[1], P[2]) exchange elements stored in array P location 1 with array P location 2. This can easily be accomplished in any programming language. A permutation algorithm will systematically call "exchange" to permute its elements.

The algorithm can also be represented as a network. The exchange operation will be implemented by an exchange module such as the one in Figure 4.2.

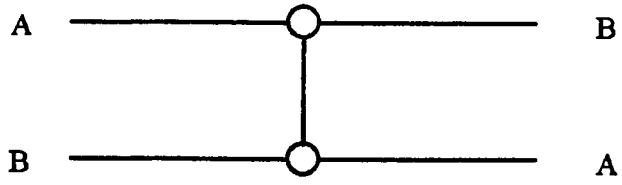


Figure 4.2. A swap module exchanges two elements.

In fact, the module in Figure 4.2 is itself a permutation network for N = 2 elements. We can build permutation networks for any value of N with the exchange module as a component. For example, Figure 4.3 is a permutation network for N = 3 elements {A, B, C}.

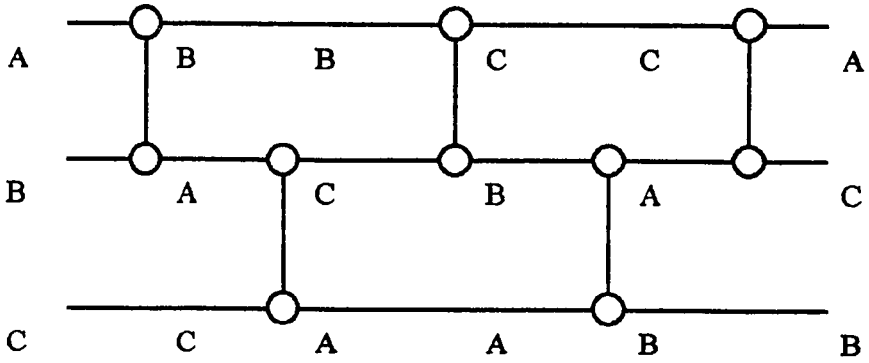


Figure 4.3 Permutation network for N = 3 elements.

This permutation network is not unique in terms of N. In fact, for N = 3, there are $3^5 = 243$ possible networks with five exchange modules. However, only 12 of them are legal in the sense that they produces sequences of distinct permutations. Figure 4.4 is a different permutation network for N = 3.

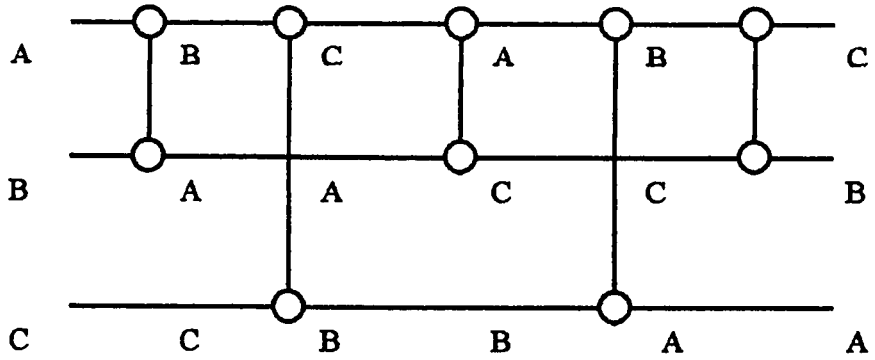


Figure 4.4 Another permutation network for N = 3 elements.

It is not hard to see that for a large N, there will be a large number of legal networks with increasing complexity. What we are interested in are methods that will systematically construct permutation networks for an arbitrary N. Furthermore, we are most interested in networks with simple enough structures such that they can be implemented on a computer efficiently.

The following two permutation networks are described in details by Sedgewick's survey of permutation generation methods [Sedgewick [18]].

Recursive Methods

From the previous section, we have seen that a permutation network for an arbitrary N can be constructed from permutation networks of N-1 elements. Furthermore, permutation networks of N-1 elements can be constructed from permutation networks of N-2 elements. Obviously, this kind of permutation network can be expressed naturally in a recursive manner.

Assuming the permutations of N elements is stored in an array P, then P[1] will be the head of the list, and P[N] will be the tail of the list. Starting off by storing the N elements in array P in any order, a recursive method to generate all permutations of the N elements repeats N times the following steps:

1. ***First generate all the permutations of the elements in P[1] . . P[N-1].***
2. ***Then, exchange P[N] with P[i] where 1 ≤ i ≤ N-1 and P[i] is different in each iteration.***

In other words, a new value is being put into P[N] in each iteration. The strategy to decide what will be the new value distinguishes various permutation methods that are based on this approach.

One way to decide the new value is to fill P[N] with the elements in decreasing order. The permutation network in Figure 4.4 operates according to this strategy with P[N] being filled with letters A, B, C in alphabetically decreasing order in each iteration. We can use this network to construct a permutation network for the letters A, B, C, D (Figure 4.5). In turn, we can use the permutation network for N elements to build permutation networks for N + 1 elements.

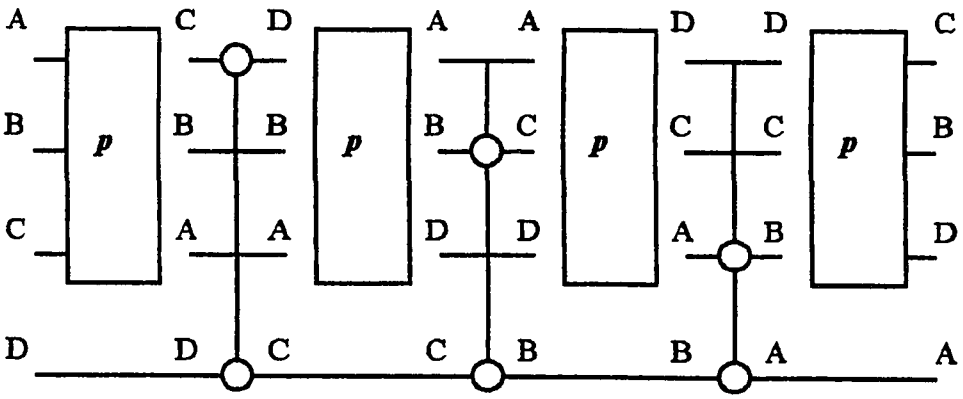


Figure 4.5: Permutation network for A, B, C, D where module P is the permutation network in Figure 4.4.

To implement this algorithm on a computer, we can define it by the following recursive function:

Algorithm 4. 1 Recursive permutation algorithm

Assume the initial permutation is store in P[1], P[2], ..., P[N].

```
Recursive_permutation(N)
{ c = 1;
  for ( ; ; ) {
    if (N > 2)
      Permutation( N - 1 );
    if c ≥ N
      break;
    else {
      exchange( P[N], P[ Find_new_element(N, c) ] );
      c = c + 1;
    }
  }; /* end for */
} /* End Recursive_permutation. */
```

Function Find_new_element in Algorithm 4.1 is the procedure that will compute the appropriate element to be exchanged with P[N]. It can be implemented in many different ways. One way to do that is to compute an index table containing the proper elements according to the values of N and c. In other words, we will precompute an index table *T* and Find_new_element will return the value T[N, c]. Because of the inherent limitation of the size of a permutation problem, N cannot be very large. Therefore, the

time it takes to compute the index table is limited and can be possibly precomputed by hand. Figure 4.6 is the index table for $N \leq 12$ adapting the rule that P[N] should be filled with the elements in decreasing order.

N												
2	1											
3	1	1										
4	1	2	3									
5	3	1	3	1								
6	3	4	3	2	3							
7	5	3	1	5	3	1						
8	5	2	7	2	1	2	3					
9	7	1	5	5	3	3	7	1				
10	7	8	1	6	5	4	9	2	3			
11	9	7	5	3	1	9	7	5	3	1		
12	9	6	3	10	9	4	3	8	9	2	3	
	1	2	3	4	5	6	7	8	9	10	11	c

Figure 4.6: Index table for T[N, c] using the rule that P[N] should be filled by the elements in decreasing order.

However, there is no reason that P[N] must be filled with elements in decreasing order. Moreover, it is not necessary to construct the index table. There are at least two simple ways to compute Find_new_element. The first one was published by M. B. Wells in 1960 [Wells [25]]. His idea is basically as follows:

```
Find_new_element(N, c)
{ if ( N is even ) and ( c > 2 )
    return( N - c )
  else
    return( N - 1 )
}
```

It is quite amazing that such a simple method would work properly. Wells gave a formal proof of this method in his paper which will be omitted here.

Another method by B. R. Heap is even simpler [Heap [9]]. The algorithm is follows:

```
Find_new_element(N, c)
{if ( N is even )
    return c;
  else
   return 1;
}
```

As we can see, a recursive approach can lead to natural and simple permutation generation algorithms. One problem with recursion is its computational cost when implemented on a computer. Recursive operations are expensive because of the multiple function calls being generated. One common way to improve efficiency is to remove the recursion by implement it as simple iteration. The general approach is to simulate the recursive call with a stack. However, removing recursion will not be our interest here and will be omitted.

Adjacent Exchanges Method

Another approach to construct a permutation network is by repeatedly exchanging adjacent elements in an existing permutation. This method is based on the idea that if we have a permutation of N-1 elements, we can generate N permutations of N elements by repeatedly inserting a new element into all possible positions of the (N-1)-permutation.

For example, if we have a permutation of 3 elements (B C D), we can generate four permutations of four elements by inserting A into the permutation (B C D) and thus create (A B C D), (B A C D), (B C A D), (B C D A).

If we stored the initial permutation as (A B C D) in an array P[1 ... 4], the insertions are equivalent to a sequence of exchanges:

exchange(P[1], P[2])

exchange(P[2], P[3])

exchange(P[3], P[4])

If we represent that as an exchange network (Figure 4.7), we can clearly see that they are just a sequence of adjacent exchanges with the first element being swept across the network.

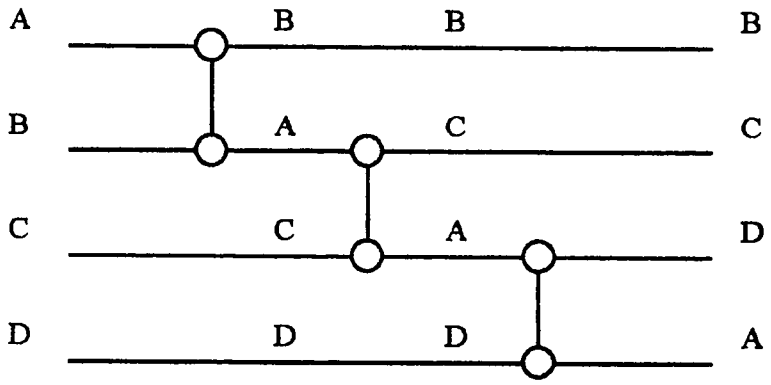
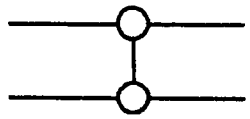
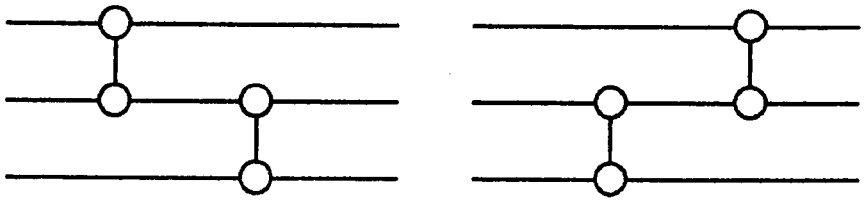


Figure 4.7: The first four exchange module for permutation of elements { A, B, C, D }.

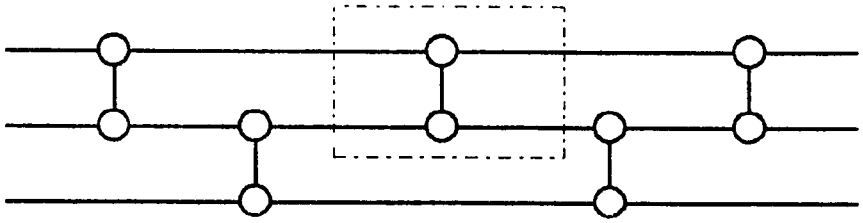
We will call the exchange modules for N elements the level-N exchange modules. Generalizing this idea, we can inductively build the permutation network for N elements using the network for N-1 elements by inserting the blocks of level-N exchange modules into the level-(N-1) exchange modules. Sound too abstract? It will be clear when we look at it graphically (Figure 4.8).



level-2 exchange module



level-3 exchange module



3 elements permutation network

Figure 4.8: Inserting level-3 exchange module into level-2 exchange module to form a three elements permutation network.

From Figure 4. 8, we can see the level-3 exchange modules are being inserted into both sides of the level-2 exchange module (enclosed in a dotted rectangle) to form a three element permutation network. This works for any N. Figure 4.9 is a four element permutation network. If we connected all the level-3 exchange modules (in dotted rectangles) together, we form a three element permutation network.

To implement this network on a computer, one obvious way is to express it recursively. However, this time let us implement it in a iterative fashion.

The adjacent exchange is easy to implement. However, how can we implement the transition between a level-N exchange to a level-(N-1) exchange? A straight forward way will be to label all the exchange modules with the level they are on. Thus, the exchange

modules of the two element permutation network will be labeled (2). The exchange modules of the three element permutation network will be labeled (3 3 2 3 3) from top to bottom of the network. The exchange modules of the four element permutation network will be labeled (4 4 4 3 4 4 4 3 4 4 4 2 4 4 4 3 4 4 4 3 4 4 4), etc. Studying these sequences carefully, we discover that for an N element permutation network, the sequence of labels are series of N-1 N's followed by a smaller label. The smaller label will be N-1 N-2 times, N-2 N-3 times and N-3 N-4 times, etc.

To write a computer program to generate this sequence, we need an array of increment counters to keep track of the number of appearances of each label. Algorithm 4.2 does exactly that, with the increment counter c[i], $2 \le i \le N$, under the condition that $1 \le c[i] \le i$.

Algorithm 4. 2 Algorithm for generating label

```
Generate_Label( )
{ for ( i == 1; i ≤ N; i++ )
    c[1] = 0;
 repeat {  i = N;
            while c[i] = i {  c[i] = 1; i --; };
            if (i > 1) {  output( i );
                          c[i] = c[i] + 1 };
        } while i > 1;
} /* End Procedure Generate_Label */
```

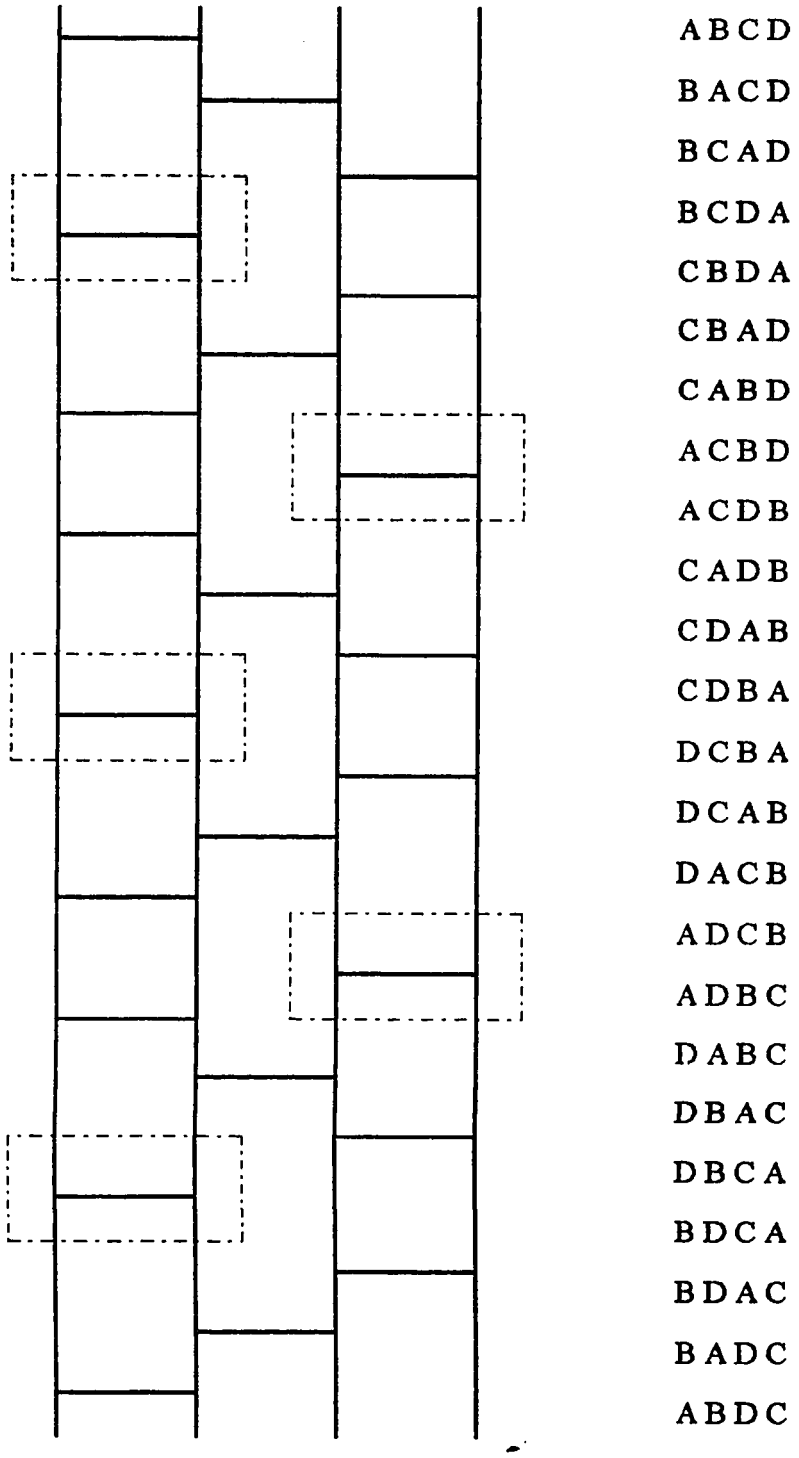


Figure 4.9: Four elements permutation network.

In Algorithm 4.2, first c[i] is initialized to be 1. Then, we output the highest index i whose counter is smaller than i. Next, we increment its counter and reset the counter of the larger index. We repeat that until i is equal to 1 again.

The second problem in implementing this network is the elements are being swept back and forth across the network. Therefore, we need an array of boolean variables to keep track of the directions of the adjacent exchanges. In our implementation, we will call the counter d[i], $2 \leq i \leq N$. They are all initialized to be TRUE.

The third problem is that the N-1 element networks are being broken up by the level-N exchange modules. There may be an offset between the exchange modules in the N-1 permutation network. Therefore, we need a variable to keep track of this offset.

Combining all these together, we have Algorithm 4. 3 which generates all the N-permutations.

Algorithm 4. 3 Adjacent exchanges permutation algorithm

```
Adjacent_Exchange_permutation(N)
{  for ( i = 1; i < N; i++) {
       c[i] = 1;
      d[i] = TRUE;
   };
  c[1] = 0;
  output(P)
  for ( ; ; ) {
   i = N;
  offset = 0;
```

```
  while ( c[i] == i ) {
     if ( ! d[i] ) offset++;
     d[i] = !d[i];
     [i] = 1;
     i --; };
   if ( i <= 1 )  break;
  else { if ( d[i] ) k = c[i] + offset;
         else  k = i - c[i] + offset;
         exchange( P[k], P[k+1]);
         output(P);
         c[i] = c[i] + 1; };
  }; /* End for */
} /* End Procedure Adjacent_exchange_permutation */
```

This algorithm looks quite different from Algorithm 4.1 because one of them uses a recursive approach while the other one uses an iterative approach. However, if we remove the recursion in Algorithm 4.1, we will see that the two algorithms are very similar. The only differences are their control structures handling the loops. In fact, the Sedgewick's paper pointed out that the two algorithms are essentially equivalent, and controlled by the same counting process that is used to generate all N-digit integers from 0 to 99 ... 999.

The fact that these two recursive and iterative algorithms are controlled by simple counting processes shows why permutations are often used as an instructive example in computer science. They illustrates nicely the relationship between counting, iteration and recursion, which are all fundamental concepts in computer science. It is useful to have such a simple example to illustrate them so well.

4. 2. 2 Suffix Reverse Algorithm

This is a relatively new algorithm discovered by Zaks [Zaks [26]]. The algorithm based on one procedure reverse(e), where the order of elements in a permutation starting from position e is reversed. In other words, a new permutation is generated from its predecessor by reversing a certain size of its suffix. The size of the suffix to be reversed is computed recursively. It turns out that the average size of this suffix is less than e ≅ 2. 8. Therefore, the running time of this algorithm is ≅ O(N!) disregarding the time to output each permutation. The time it takes to generate each permutation on average is constant.

First, let us describe how the size of the suffix is computed. We will denote the sequence of sizes of the suffix to be reversed in generating a permutation for N elements to bc S_N. S_N is defined recursively as follows:

$S_2 = 2.$

$S_N = (S_{N-1}N)^{N-1}S_{N-1}$, $N > 2.$

For example, S_3 = 23232, S_4 = 2323242323242323242323 and

S_5 =2323242323242323242323252323242323242323242323325. . . .

With S_N defined, the algorithm is presented as Algorithm 4.4.

Algorithm 4. 4 Reverse suffix algorithm

```
Reverse_suffix_permutation(N)
{  current_perm[1 ... N] = [1, 2, 3, 4, ..., N];
       Compute_size(N);
   for (i == 1, i≤N, i++)
      Reverse(S_N[i]);
   Output(current_perm);
} /* End procedure Reverse_suffix_permutation.  */
```

Procedure Compute_size can be implemented recursively as defined. Procedure Reverse(e) just reverses the order of elements of array current_perm starting from position e. It can be easily implemented in most computer languages as follow:

```
Procedure reverse(e)
{ temp[1 ... N];
  for (i = e; i≤N; i++)
    temp[i] = current_perm[i];
  for (i == N; i≥e; i--)
    current_perm[i] = temp[e + N - i]
}
```

Using induction, we can easily prove that starting with the permutation (1 2 3 ... N) and applying the sequence S_N of suffix reversals, this algorithm generates all N! permutations of N elements.

Each call to reverse will take running time e, and e is each entry from the sequence S_N. It is not difficult to observe that N appears in S_N exactly N!/(N-1)! times, (N-1) appears in the positions that are multiples of (N-2)! but not of (N-1)! and thus appears N!/(N-2)! - N!/(N-1)! times. In general, the number i, $2 \le i \le N$, appears in S_N (N!/(i-1)! - N!/i!) times. Therefore, the expected suffix size is:

$$1/N! * (N * (N! /(N-1)!) + \sum_{i=2}^{N-1} i * (N!/(i-1)! - N!/i!) = \sum_{i=0}^{N-1} 1/i! < 2.8$$

Therefore, procedure Reverse has an average running time of $e \cong 2.8 = O(1)$. Procedure Compute_size can easily be implemented recursively. However, we can generate the sequence S_N more efficiently by using iterative methods. We can see each i, $2 \le i \le N$, appears in a fixed pattern. We can use a tactic similar to the one we used to

generate labels in the adjacent exchanges permutation network. Using an increment counter array to count the number of appearances of each i, we can generate the sequence S_N in linear time. Therefore, the main cost of this algorithm is still the loop that calls procedure reverse N! times. Each output takes O(N) time since there are N elements. Overall, this algorithm has running time O(N! * N).

4. 3 Parallel Permutation Algorithm

4. 3. 1 Parallel Algorithm Using Linear Processors Array

The first parallel permutation algorithm [Chen and Chern [5]] presented in this thesis not only generates permutations for m out of N objects, it generates all the permutations of at most m out of N objects. For example, all the permutations of at most 3 out of 3 objects are:

{ (1), (2), (3), (1 2), (2 1), (1 3), (3 1), (2 3), (3 2), (1 2 3), (1 3 2), (2 3 1), (2 1 3), (3 2 1), (3 1 2) }

The architecture being used in this algorithm consists of a linear array of k processing elements and a device called selector. The processing elements are arranged in a ring structure as in Figure 4.10.

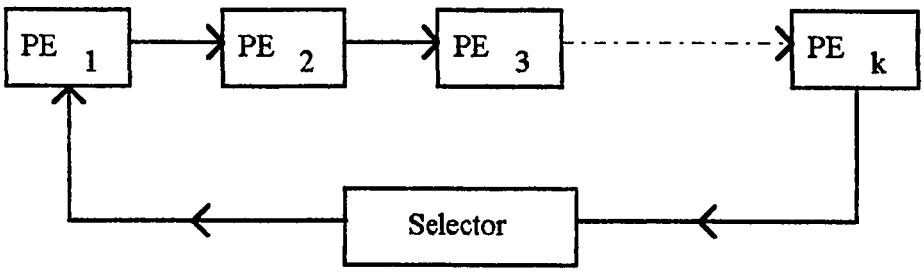


Figure 4.10 : Linear processors array and selector.

Each processing element maintains a stack of at most N objects. At each step, a cyclic right shift is performed. In other words, the top element of each processing element's stack is transferred to its right neighbor. The only exception is the selector. It receives an element x from processing element PE_k and sends an element y to processing element PE_1. Element x is just the top element on processing element PE_k's stack. Element y is determined as follow:

if $x > k$

$y = x - k$

else

$y = N + x - k$

If we considered that the objects forms a circular list (1 2 3 4 ... N), y is k positions from x. The basic ideas behind this algorithm are:

1) All the permutations of N objects $\{I_1, I_2, I_3, ..., I_N\}$ can be partitioned into N sets. Each set with a different leading element I_i, $1 \le i \le N$. And each set can be generated independently.

2) If $S = \{s_1, s_2, s_3, ..., s_{N!}\}$ is the set of all permutations of N objects $\{I_1, I_2, I_3, ..., I_N\}$, we can generate all the permutations of N+1 objects $\{I_1, I_2, I_3, ..., I_N, I_{new}\}$ with leading element I_{new} by adding I_{new} to each permutation of S, i. e. ,

$S' = \{I_{new}s_1, I_{new}s_2, I_{new}s_3, ..., I_{new}s_{N!}\}$.

Based on the first idea, if the time it takes to generate all the permutations with the same leading element is T, given k processors, we can generate all the permutations in $T * \lceil N / k \rceil$ time.

We can use the second idea to generate each permutation recursively. Each permutation of N objects is generated by appending a distinct object to permutations of N-1 objects. And each permutation of N-1 objects is generated by appending a distinct object to permutations of N-2 objects, etc. In this algorithm, the distinct objects are chosen from the objects being transferred by the cyclic right shifts. Finally, the algorithm is presented as Algorithm 4. 5. Figure 4.11 shows an example of N = 4, m = 3 and k = 3.

Algorithm 4. 5 Parallel linear array permutation algorithm

```
Parallel_linear_array_permutation(m, N)
  Stack S_i, 1 ≤ i ≤ k
  Top_i, /* Top element of stack S_i */
  counter   /* counting the number of shifting on the same level */
{ for each processing element PE_i,  1 ≤ i ≤ k, {
    push i; counter = 0; send Top_i;
   While (!terminate) {
     receive element shifted, e_i
     if (e_i is distinct from other elements on the stack) {
       push e_i; counter = 0; output S_i ; }
     else {  if (counter = N - 1)   pop S_i;
              if (the first element of the stack was popped)
                if (the first element is bigger than m)  terminate
                else  re-initialize the stack with i+k, i+k ≤ N };
     send Top_i;
  } /* End while */ } /* End for */
} /* End procedure Parallel_linear_array_permutation */
```

** mark time step where permutations were generated*

time step	P1	P2	P3	time step	P1	P2	P3
1 *	1	2	3	15	4		
2 *	4	1	2	16 *	3		
	1	2	3		3		
	3	4	1		2		
3 *	4	1	2	17 *	3		
	1	2	3		4		
	2	3	4		1		
4 *	4	1	2	18 *	3		
	1	2	3		4		
	1	2	3	19 *	2		
5	4	1	2		4		
	1	2	3				
6 *	3	4	1		1		
	1	2	3	20 *	2		
					4		
	2	3	4		4		
7 *	3	4	1	21	2		
	1	2	3		4		
	1	2	3		3		
8	3	4	1	22 *	2		
	1	2	3		4		
	4	1	2	23*	1		
9 *	3	4	1		4		
	1	2	3				
10 *	2	3	4		4		
	1	2	3	24	1		
					4		
	1	2	3		3		
11	2	3	4	25 *	1		
	1	2	3		4		
	4	1	2		2		
12 *	2	3	4	26 *	1		
	1	2	3		4		
	3	4	1	27	2		
13 *	2	3	4		4		
	1	2	3				
14	1	2	3	28	4		

Figure 4.11: Example with N = 4, m = 3, k = 3.

Now, we will analyze the time complexity of the algorithm. If each stack maintains an array of size N to keep track of the elements that are already on the stack, we can check if an element is distinct from other elements on the stack in constant time. Outputting a permutation of size m takes linear time (O(m)). All other operations in each iteration clearly can be performed in constant time. Therefore, the major concern is how many iterations it takes to generate all the desired permutations.

Let T_{valid} = Number of i-permutations from N objects starting with the same leading elements, where $1 \leq 3 \leq m$.

$$T_{valid} = \sum_{i=1}^{m} {}^{N-1}P_i \text{, recall that } {}^{N}P_m = N! / (N - m)!$$

However, there are some instances where the permutations are not valid. Looking back at the example, at each iteration of the 2nd level of the stack, there are two elements at the 3rd level of the stack which do not apply to a valid permutation. There are three elements which apply to a valid permutation at the 2nd level. As a result, (3 * 2) iterations are wasted. In fact, iterating at each level i of the stack, the number of iterations wasted is

$$\left(\prod_{j=2}^{i} N-1-j \right) * j - 1.$$

There are m stack levels. Therefore, the total number of time steps wasted, T_{wasted}, is

$$T_{wasted} = \sum_{i=1}^{m} \left(\prod_{j=1}^{i} N-1-j \right) * j - 1.$$

Every valid permutation takes linear time to output. Using k processing elements, it takes $\lceil N / k \rceil$ iterations of the algorithm to generate all the desired permutations. The overall running time of the algorithm $T_{overall}$ is

$$T_{overall} = \lceil N/k \rceil * T_{valid} * O(m) + T_{wasted}$$

It is obvious that $T_{wasted} < (T_{valid} * O(m))$. Therefore,

$$T_{overall} = \lceil N/k \rceil * O(T_{valid} * O(m)).$$

There are k processing elements, therefore, the total computational cost T_{total} is

$$T_{total} = T_{overall} * k = N * O(T_{valid} * O(m))$$

Again, T_{valid} is the number of valid permutations starting with the same leading elements. There are N elements. Therefore, overall, there are $(N * T_{valid})$ permutations. Since there are $(N * T_{valid})$ permutations and each of them has size O(m), any optimal sequential algorithm must have time complexity greater than or equal to $N * O(T_{valid} * O(m)) = T_{total}$. By the law of linear speedup, this parallel algorithm is optimal because its total parallel running time is equal to the optimal sequential time.

Even though this parallel algorithm is mathematically optimal, there are still a few drawbacks.

First, the time steps wasted T_{wasted} can be significant. Fortunately, the time steps involving invalid elements are detected and eliminated at the first step of the iteration. Therefore, the operational times for those time steps are very small.

Secondly, if N is not a multiple of k, as is the case in our example in Figure 4.11, some of the processing elements will be left idle at the final iteration of the algorithm. In some cases, the portion of processing elements left idle compared with the number of active processing elements can be large.

Thirdly, most of the applications only need to generate permutations for a specific m. An algorithm generating all permutations of at most m may not be so practical.

Fortunately, in this case, permutations generated with less than m elements are also used in generating larger permutations. Therefore, not much time is wasted. In fact, this is the reason why this algorithm can achieve the optimal efficiency. Normally, in an algorithm generating permutations for m out of N objects, the time spent building permutations with less than m elements will be calculated as part of the effort to generate each m-permutation. However, in this case, permutations with less than m elements are also part of the goal. This allows the time complexity of this algorithm to be comparable to an optimal sequential algorithm that generates all the at most m permutations.

Finally, the permutations generated in this algorithm aren't in any specified order. In fact, permutations with different numbers of elements are all mixed together. It would be difficult to access any particular permutation. For example, if we only want 2-permutations and 4-permutations out of four elements, we will still need to generate all the permutations of three elements.

In the next section, using the same basic idea, we will design an algorithm that generates all N-permutation of N elements. And we will take a closer look at the implementation details.

4. 3. 2 Modified Version of Algorithm 4.5

In many applications, it is only necessary to generate all N! permutations of N objects. By relaxing this restriction, we can design an algorithm with simpler architecture and more efficient implementation.

Unlike Algorithm 4. 5, where the number of processing elements being used is independent of the number of objects, in this algorithm, we will use N processing elements

to generate all the N-permutations. In other words, the number of processing elements used will grow as the number of objects increases.

By restricting the number of processing elements being used, we are able to eliminate the use of a selector. This contributes to the simpler architecture in Figure 4.12.

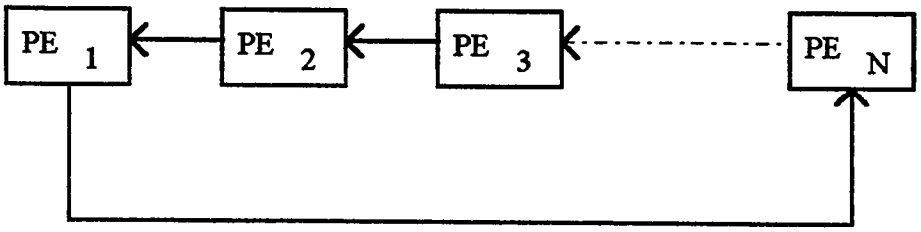


Figure 4.12 : Circular linear processor array.

As in Algorithm 4. 5, each processing element will maintain a stack of at most N elements. At the beginning, each stack of processing elements, PE_i, will be initialized with object i. Each processing element will keep track of the number of elements and the top element in its stack.

At each iteration, each processing element does a cyclic left shift of the top elements of its stack. After each processing element has received its new element, it outputs a new permutation if it is available. Then, it pushes or pops its stack depending on the element it had received and the elements on its stack. As in Algorithm 4.5, each processing element PE_i will produce all the permutations with leading element i. It does that by cycling through each possible object on its stack systematically. Every time a distinct object is found, the stack is expanded with this object. When there are N distinct objects on the stack, a permutation is successfully generated. When it has cycled through all the objects on the second level of its stack, then all the permutations have been generated, and the algorithm terminates.

Before we state the algorithm, we will list some of the data structures used in the algorithm. First, there is the stack $\mathbf{Stack_i}$ maintained by each processing element PE_i. Next, to determine if a distinct object is received, an array of N elements **Used** is maintained by each processing element to record the current objects that are on the stack. **Used** will be a boolean array of N bits, every time a new object is added to the stack, the corresponding bit will be set. When an element is released from the stack (when the stack is popped), that bit will be cleared.

In each level of the stack, the algorithm cycles through all the objects in searching for a distinct object. If no distinct object is found after all the N objects are examined, it means all the possible objects have been examined at that level and the stack will be popped. In Algorithm 4.5, this is accomplished by iterating the algorithm N times on each level of the stack. In our algorithm, instead, we will do this by using an array **Initial_element**. Initial_element is an array of N elements, every time a new object is push onto level k of the stack, Initial_element[k] will be assigned the new object. Now, we can compare the object received against Initial_element[k] to determine if all the N objects have been examined.

Each time the processing elements receive a new object from their neighbors, they may push that object onto the stack, incrementing the number of elements on the stack. However, when it is cycling through the N objects to search for a new distinct object, instead of incrementing the stack, it will replace the current top element with the new object. We need to find some way to distinguish them. It turns out that we want to replace the current top element with a new object only when we are searching for a new distinct object. After a distinct object is found, the next object received will be pushed onto the stack. In other words, we only want to increment the stack at the beginning of the algorithm and when the stack has just been popped. Therefore, we can use a boolean

variable **Popped**. **Popped** is clear at the beginning of the algorithm. Every time the stack is popped, **Popped** will be set and the next object received will replace the top element instead of being added onto the top of the stack. When a distinct object is found, Popped will be cleared again and the new object received will be pushed onto the stack. Finally, the algorithm is presented as Algorithm 4. 6.

Algorithm 4. 6 Modified parallel linear array permutation algorithm

```
Parallel_linear_array_permutationII(N)
{ Counter = 1; Stack_i[Counter] = i;
  Initial_element[Counter] = i;
  Used = 0; Popped = 0;
  While not terminate {
    received New_object
    if ( Counter < N ) and ( not Popped ) {
      Counter ++;
      Stack_i[Counter] = Initial_element[Counter] = New_object;
      if (Counter == N) and ( not Used[New_object]) {
        Output_Permutation;
        Counter --;
        Used[ Stack[Counter] ] = 0;
        Popped = 1;
      }
      else if (( Counter < N ) and ( Used[New_object] ))  Popped = 1;
      else Used[New_object] = 1;
    }  /* End if */
```

```
    else  if (Counter == N ) and  ( not Popped )
      if (Initial_element[Counter] ==  New_object) {
        Counter --;
        Popped = 1; };
      else {
        stack[Counter] = New_object;
        if ( not Used[New_object])
          Output_Permutation;
          Counter --;
          Used[ Stack[Counter] ] = 0;
          Popped =1; };
    else
      if (Initial_element[Counter] == New_object
        if (Counter == 2) terminate;
        else {
          Counter --;
          Used[ Stack[Counter] ] = 0;
          Popped = 1; };
      else {
        if ( not Used[New_object] ) {
            used[New_object] = 1; Popped = 0; };
        Stack[Counter] = New_object; };
  send Stack[Counter]
  }  /* End while */
} /* End procedure Parallel_linear_array_permutationII */
```

Algorithm 4.6 has similar time complexity to Algorithm 4. 5. Each processing element produces (N-1)! permutations. To generate each permutation, the processing element needs to go through the algorithm at least N times. Therefore, the number of iterations taken at each processing element is

$$(N-1)! * N + \textit{time wasted on invalid permutation} = N! + T_{wasted}$$

The time wasted on invalid permutation is the same as Algorithm 4. 5, which is

$$T_{wasted} = \sum_{i=1}^{N-1} \left(\prod_{j=1}^{i} N-1-j \right) * j - 1.$$

It is not difficult to see that $T_{wasted} \leq N!$. Therefore, the number of iterations taken at each processing element is O((N-1!) * N). There are N number of processing elements, therefore, the total computation time is

$$T_{total} = N * O(N!).$$

There are N! number of permutations and each of them has size N. As a result, the optimal sequential algorithm must have time complexity greater than or equal to N! * N. Since the total computation time of Algorithm 4.6 matches the optimal sequential time, by the law of linear speed up, Algorithm 4.6 is cost optimal.

4. 3. 3 Minimal Change Order Algorithm

Like the sequential algorithms presented in section 4.2.1, one of the simplest ways to generate a new permutation from an existing one is to exchange two elements from the existing permutation. In other words, two successive permutations will only differ in two positions - the smallest difference possible. The permutations generated under this condition are said to be generated in minimal change order. A permutation algorithm complying with this rule is called a minimal change algorithm.

As we have discussed in section 4.2.1, algorithms based on this idea differ only in their way of choosing the two elements to be exchanged. One approach presented by Johnson [Sedgewick [18]] is to exchange two elements that are adjacent to each other. In what follows, we will present an algorithm proposed by Tsay and Lee [Tsay & Lee [24]] that modified and parallelized Johnson's method to obtain an optimal parallel algorithm that generates permutations in minimal change order.

First, we will review Johnson's method. His method utilizes the ideas of an inversion of a permutation and an interchanging operation. We will explain the two notations in the follows.

Inversion of a permutation I(k)

Let $P = (p_1, p_2, ..., p_a, ..., p_b, ..., p_N)$ be a permutation of N objects. The pair p_a and p_b is an inversion of P if $p_a > p_b$. For Johnson's method, we will define a function I(k) for $1 \le k \le N$. I(k) is 1 if the number of inversions on elements 1, 2, 3, ..., k-1 is odd. Otherwise, I(k) is 0. Moreover, I(1) = I(2) = 0.

For example, let P = (4 1 3 2), there are four inversions, namely, (4 1), (4 3), (4 2) and (3 2). I(1) = I(2) = I(3) = 0. I(4) is 1 because, with respect to element 4, there is an inversion (3 2).

Interchanging operation $T(p_k)$

For a permutation and an element p_k, interchanging operation $T(p_k)$ is defined under the following three cases.

Case 1 : If $I(p_k) = 0$ and the element p_k has an immediate left neighbor p_{k-1}, p_k is exchanged with p_{k-1} if $p_k > p_{k-1}$.

Case 2 : If $I(p_k) = 1$ and the element p_k has an immediate right neighbor p_{k+1}, p_k is exchanged with p_{k+1} if $p_k > p_{k+1}$.

case 3 : If the permutation does not satisfy either case 1 or case 2, $T(p_k)$ is undefined, so no action is taken.

For the previous example, P = (4 1 3 2). After we apply T(4) to P, P become (1 4 3 2) since I(4) = 1 and it has a smaller immediate right neighbor.

With the two functions defined, Johnson's method is follows:

(1) Initialize the algorithm with any permutation P.

(2) Repeat the following N! times:

(i) Apply T(m) to P where m is the largest element for which T(m) is defined.

(ii) Complement the result of I(k) for $m < k \leq N$.

For example, using Johnson's method, we can generate all the 4-permutations starting with (1 2 3 4) as shown in Figure 4.13.

k	$Perm_k$	T(m)	k mod 4		k	$Perm_k$	T(m)	k mod 4
0	(1 2 3 4)	T(4)	0		12	(4 3 2 1)	T(4)	0
1	(1 2 4 3)	T(4)	1		13	(3 4 2 1)	T(4)	1
2	(1 4 2 3)	T(4)	2		14	(3 2 4 1)	T(4)	2
3	(4 1 2 3)	T(3)	3		15	(3 2 1 4)	T(3)	3
4	(4 1 3 2)	T(4)	0		16	(2 3 1 4)	T(4)	0
5	(1 4 3 2)	T(4)	1		17	(2 3 4 1)	T(4)	1
6	(1 3 4 2)	T(4)	2		18	(2 4 3 1)	T(4)	2
7	(1 3 2 4)	T(3)	3		19	(4 2 3 1)	T(3)	3
8	(3 1 2 4)	T(4)	0		20	(4 2 1 3)	T(4)	0
9	(3 1 4 2)	T(4)	1		21	(2 4 1 3)	T(4)	1
10	(3 4 1 2)	T(4)	2		22	(2 1 4 3)	T(4)	2
11	(4 3 1 2)	T(2)	3		23	(2 1 3 4)		3

Figure 4. 13 Permutation generated by Johnson's method.

Figure 4.13 shown all the 4-permutations $Perm_k$ at each iteration k, $0 \le k \le (4! - 1)$, of the algorithm. It also shows the interchanging operation applied in each step and the value (k mod 4) in each step. By observing Figure 4.13, we can see Johnson's method always applies T(4) to the permutation P_k when (k mod 4) is not equal to 3. In fact, if we define the set S_i,

$$S_i = \{s \mid (s \bmod N) = i,\ 0 \le s \le N! - 2\}\ \forall i,\ 0 \le i \le N! - 1,$$

and initialize the Johnson's method with $Perm_0$ = (1 2 3 ... N), interchanging operation T(N) is always applied to $Perm_k$ when $k \notin S_{N-1}$, $0 \le k \le N! - 2$. Now, all that left is how the interchange operation T(m) will be applied in the cases that $k \in S_{N-1}$.

Assume $k \in S_0$, i. e. k is divisible by N. Permutations $Perm_k$, $Perm_{k+1}$, ..., $Perm_{k+N-1}$ are called a permutation block. Clearly, the last permutation $Perm_{k+N-1}$ of a permutation block can be obtained by applying T(N) N-1 times to $Perm_k$, the first permutation of a permutation block. In other words, given the first permutation of a permutation block, we can compute the last permutation of a permutation block. Now, we will define how T(m) will be applied in the case that $k \in S_{N-1}$.

Suppose $Perm_k = (p_{k1}\ p_{k2} \cdots p_{kn})$ is the first permutation of a permutation block, interchanging operation T(m) will be applied to the last permutation of the permutation block with m define as

$$m = Max\{i \mid T(i) \text{ is defined in } Perm_k \text{ and } i \ne N,\ 1 \le i \le N\}.$$

Finally, Tsay and Lee's version of Johnson's method is presented as Algorithm 4.7.

Algorithm 4. 7 Tsay and Lee's minimal change algorithm

```
Procedure Minimal_change_permutation(N)
 int current_perm[N] = {1, 2, 3, . . . . . , N}; /* Permutation generated at each step */
 int inverse[N] = {0};
 int m = N-1;
{ Output(current_perm);
  While ( m != 0 ) {
    for (counter = 1; counter ≤ (N - 1); counter ++) {
      T(N);
      Output(current_perm); };
    T(m);
    Output(current_perm);
    /* Complement I(k), m ≤ k ≤ N.  */
    for (counter = m; counter ≤ N; counter++)
     inverse[counter] = ~inverse[counter];
/* m = Max{ {0} ∪ { current_perm[i] | T(current_perm[i]) is defined, 1 ≤ i ≤ N }*/
    m = 0;
    for (counter = 1; counter ≤ N; counter ++)
      if (current_perm[counter] ≠ N)
       if ( is_define_in_T( current_perm[counter] ) )
         if (current_perm[counter] > m)
           m = current_perm[counter];
 }  /* End while */
} /* End procedure minimal_change_permutation */
```

Next we will try to parallelize Algorithm 4.7. The parallel algorithm will run on a linear array of N processing elements. Each processing element PE_i maintains four memory locations, current_perm[i], inverse[i], m[i], counter[i]. Memory locations current_perm[i] and inverse[i] are the same as defined in Algorithm 4.7. Memory location m[i] is the value being applied with interchanging operation T(m) in association with processing element PE_i. Memory location counter[i] is the counter being used by processing element PE_i. The parallel algorithm basically tries to parallelize each step of Algorithm 4.7 and is presented as Algorithm 4.8.

Algorithm 4.8 Parallel Tsay and Lee's minimal change algorithm

```
Procedure Parallel_minimal_change_permutation(N)
{m[0] = m[N+1] = 0;
 for ( i = 1; i ≤ N; i++)
   for each processing element PE_i do in parallel {
    /* Step one : Initializations */
    current_perm[i] = i; Inverse[i] = 0; m[i] = N - 1; counter[i] = 0; Output(current_perm);
    while ( not Terminate ) {
     for ( counter[i] = 0; counter[i] < N-1; counter[i]++)
   /* Step two: Apply T(N) to current_perm, where current_perm is the
    kth permutation and k ∉ S_{N-1}. */
      if ( (current_perm[i] == N) and ( inverse[i] == 0) ) {
        exchange(current_perm[i], current_perm[i-1]);
        exchange(inverse[i], inverse[i-1]);
      } else if ( (current_perm[i] = N) and ( inverse[i] == 1) ) {
        exchange(current_perm[i], current_perm[i+1]);
```

```
      exchange(inverse[i], inverse[i+1]); };
    Output(current_perm);
    /* Step 2. 1 */ m[i] = max(m[i-1], m[i], m[i+1]);  } /* End for */
    /* Step three : Apply T(m) to current_perm, where current_perm is the
       kth permutation and k ∈ S_{N-1} */
    if ( (current_perm[i] == m[i]) and (inverse[i] == 0) ) {
      exchange(current_perm[i], current_perm[i-1]);
      exchange(inverse[i]; inverse[i-1]);
    } else if ( (current_perm[i] == m[i]) and (inverse[i] == 1) ) {
      exchange(current_perm[i], current_perm[i+1]);
      exchange(inverse[i]; inverse[i+1]); };
    else if ( m[i] == 0 ) Terminate;
    Output(current_perm);
    /* Step four : Complement inverse(i), m < i ≤ m.  */
    if ( current_perm[i] > m[i] ) inverse[i] = ~inverse[i];
    /* Step five : Initialize m[i] */
    if ( current_perm[i] == N) m[i] = 0;
    if ( (inverse[i] == 0) and (i > 1) and (current_perm[i-1] < current_perm[i]) )
      m[i] = current_perm[i];
    else if ((inverse[i] == 0) and (i > 1) and (current_perm[i-1] < current_perm[i]))
      m[i] = current_perm[i];
    else  m[i] = 0; }  /* End for */
  } /* End while */
 } /* End for */
} /* End procedure parallel_minimal_change_permutation */
```

Algorithm 4.8 is actually very similar to Algorithm 4.7 except the initialization phase and the computation of the value m in Algorithm 4.8 are done in parallel. To compute m in parallel, first m is initialized for each processing element in step five. The idea is if T(current_perm[i]) is defined, m[i] is initialized to current_perm[i]. Otherwise, it is initialized to 0. Since m cannot be equal to N, if current_perm[i] is equal to N, it should also be initialized to 0. Then, the actual value m to be applied with interchanging operation T(m) is computed in statement 2. 1 " **m[i] = max(m[i-1], m[i], m[i+1])** ". There seems to be a problem with this statement on the surface. This statement is executed by all the N processing elements at the same time, and in the statement, there is an access to memory locations of neighboring processing elements, namely, m[i-1] and m[i+1]. However, when this statement is being executed on one processing element, the values stored in m[i-1] and m[i+1] could be changed by processing elements P_{i-1} and P_{i+1} when they execute the same statement. It seems that there is a synchronization problem in here. Will the order of executions of this statement by the processing elements affect the final result? Fortunately, the answer is no. This is because the statement returns the maximum value in m between a processing element and its two immediate neighbors. Since this statement is executed by each processing element N - 1 times by all the N processing elements, the maximum m among all of the processing elements will be distributed to each processing element at the end of the execution of this statement. By the time the value m is being accessed at step three, each processing element will always have the same value m in its corresponding memory locations.

Each statement of Algorithm 4.8 can be executed in constant time. The algorithm is iterated for each permutation block, and each permutation block consists of N permutations. As a result, the algorithm will be executed (N! / N) times and each iteration is O(N). Therefore, the algorithm has a running time of O(N!). There are N processing

elements. Thus, the total computation cost of the algorithm is O(N!*N) which matches the optimal running time for a sequential algorithm. Therefore, Algorithm 4.8 is cost optimal.

One last observation is the algorithm can be started with any initial permutation. However, it is most beneficial to start the algorithm with permutation (1 2 3 ... N). This is because in this case we can initialize inverse[1 ... N] to be zero. Otherwise, we need to compute the function I(k), $1 \leq k \leq N$, which is an $O(N^2)$ function. Even though in the analysis of time complexity, it will still be dominated by the O(N-1)! loop, elimination of computing the function I(k) reduces the constant of the time complexity.

4. 3. 4 Systolic Algorithm

The last algorithm we are going to discuss in this chapter generates permutations by using a systolic approach. It was introduced by Lin in 1989 [Lin[12]]. A typical systolic array consists of a linear array of processing elements connected by communication links. Two neighboring processing elements communicate by sending data through the communication links. When processing element PE_i sends some data to PE_{i+1} through a communication link (let's call it the k-link), we say that k-link is an output link for PE_i and an input link for PE_{i+1}. We will denote the data being sent from PE_i as k_{out}, and the data received by PE_{i+1} as k_{in}. As the data being pumped through the processing elements regularly and synchronously, each processing element performs the following three operations:

(1) Read the data from its input links.

(2) Execute one iteration of the systolic algorithm.

(3) Send the data to its output links.

The time it takes to perform the above three operations is called a time step. In this section, we will use the notation PE(i, t) to describe a processing element. The value i

refers to the index associated with a processing element. The value t refers to the time step. For example, PE(3, 5) corresponds to the processing element PE_3 at time step 5. This systolic algorithm generates all the N-permutations using N processing elements. One permutation will be generated in each time step. Therefore, we will denote a processing element by PE(i, t), $1 \leq i \leq N$ and $1 \leq t \leq N!$.

Since the systolic algorithm generates an output in each time step, it reduces the time to generate an output from N time steps sequentially to one time step systolically. Clearly, the systolic algorithm being executed at each time step is the core of the function of a systolic array. Next, we will discuss the systolic version of Lin's permutation algorithm.

First, we will assume the N objects are integers 1, 2, 3, ..., N. The idea behind this algorithm is to first chose any (N-1)-permutation P_A. By appending the element N to P_A, we obtained an N-permutation P_B. We will follow Lin's notation and call this permutation a header. Now, we will apply a modular operation to the header P_B (N-1) times to produce the next (N-1) N-permutations. The modular operation is for any integer j, $1 \leq j \leq N-1$, we add j to each element of the header P_B under modulo base N to produce a new N-permutation. For example, if the header P_B is (1 2 3 4 5), the N-1 permutations produced from P_B are (2 3 4 5 1), (3 4 5 1 2), (4 5 1 2 3) and (5 1 2 3 4). Again using Lin's notation, we will call the above process of applying the modular operation N-1 times on header P_B a cycle-N-perms with header P_B. Therefore, this systolic algorithm is basically applying the cycle-N-perms with each of the (N-1)! different headers. Moreover, during the execution of the cycle-N-perm with a header, we must compute the next header in time in order to perform another cycle-N-perm continuously. All of these contribute to the following design considerations:

(1) Each processing element will handle one position of each permutation. N permutations will be produced in each cycle-N-perm, each processing element will maintain a counter K to count the number of permutations produced within each cycle.

(2) A c-link going from PE(i, t) to PE(i, t+1) denotes the transition of the ith position of the permutation from one time step to the next. This value is output as part of the permutation and is stored for computing the next value.

(3) A d-link going from PE(i, t) to PE(i-1, t+1) is used to compute the ith position of the next header which is temporarily stored at register T. It does that by applying modular operations on T with the number of operations stored at register Q and the module bases storing at register S. In each time step, if $d_{in} > 0$, it stores d_{in} in Q and N-1 in S. Then, it applies the modular operations on T Q times with module base varying from N-1, N-2, N-3, ..., N-d_{in} in each operation.

(4) An e-link going from PE(i, t) to PE(i-1, t+1) transmits a flag F to its left neighbor indicating that PE(i, t) has sent prepared information for the action of the next header. In each time step, if (e_{in} = 1 and T = i and F = 0), then PE_i sends $d_{out} = d_{in}+1$ and e_{in} = 1 to its left neighbor and sets the flag F. Otherwise, it sends $d_{out} = d_{in}$ and e_{out} = 0 to its left neighbor.

(5) Register K is being used to count the number of permutations produced in each cycle-N-perm. When K reaches N, the new header stored in T is output as c_{out}. In any other case, the permutation produced by the cycle-N-perm is output as c_{out}.

With all the design considerations in mind, the systolic algorithm introduced by Lin is presented as Algorithm 4.9 which utilizes the systolic array presented in Figure 4.14.

Algorithm 4. 9 Systolic algorithm for generating permutation

```
Systolic_Permutation(N)
{  int K, T, Q, S, F, c_in, c_out, d_in, d_out, e_in, e_out;
   for ( i = 1; i ≤ N; i++)
    for all processing element PE_i do in parallel {
      /* Initialization Phrase */
      K = N; T = i; S = Q = F = 0; c_in = c_out = d_in = d_out = e_in = e_out = 0;
      /* All the links are initialized to zero except e_in for PE_N is 1.  */
       if ( i = = N ) e_in = 1;
      /* Execution phrase */
      do {  if ( K < N ) Modulo_c_out( );
            else New_header_c_out( );
            if ( d_in > 0 ) Evaluate_new_header( );
           if (( e_in = 1 ) and ( T = i ) and ( F = 0 )) Signal_d_out( );
           else Transmit_d_value( );
      } while e_out ≠ 1 for PE_1;
} /* End procedure Systolic_Permutation */
```

The five procedures are defined as follows:

```
Modulo_c_out( )
{ if ( c_in < N )  c_out = c_in + 1;
   else c_out = 1;
   K++;
} /* End procedure Modulo_c_out */
```

```
New_header_c_out( )
{  c_out = T;
   K = 1;  F = 0;
} /* End procedure New_header_c_out.  */

Evaluate_new_header( )
{  Q = d_in;
   S = N - 1;
   while ( Q > 0 ) {
    if ( T < S ) T++;
    else  T = 1;
    Q--;
    S--:  } /* End while */
} /* End procedure Evaluate_new_header.  */

Signal_d_out( )
{  d_out = d_in + 1;
   e_out = 1;
   F = 1;
} /* End procedure Signal_d_out.  */

Transmit_d_value( )
{ d_out = d_in;
  e_out = 0;
} /* End procedure Transmit_d_value.  */
```

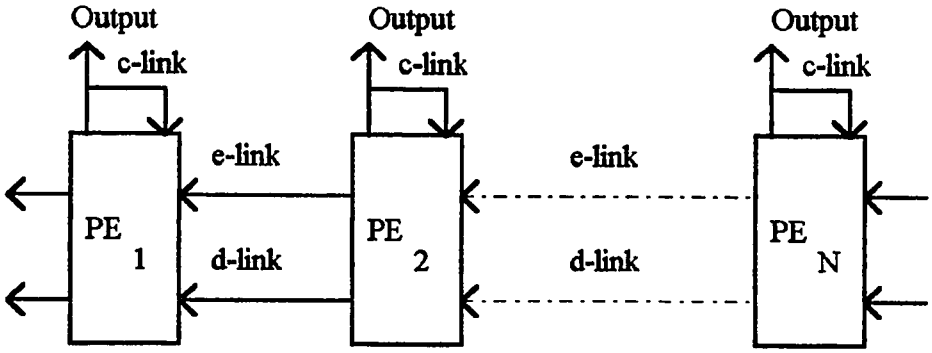


Figure 4.14: Systolic array used in Algorithm 4.9.

Each time step of Algorithm 4. 9 consists of calling the five procedures sequentially. Procedures Modulo_c_out, New_header_c_out, Signal_d_value and Transmit_d_value are all constant time procedures. The only iterations are in procedure Evaluate_new_header where a while loop is iterated d_{in} times. The value of d_{in} varies from 0 to N - 1. Even though in most cases d_{in} is equal to zero or one, in the worst case d_{in} is equal to N - 1. That would normally imply that procedure Evaluate_new_header has O(N) time complexity. However, we are evaluating Algorithm 4.9 on a time step to time step and per processing element basis. In each time step, procedure Evaluate_new_header is called only once among N processing elements. Therefore, in our analysis, we maintain that in a single time step procedure Evaluate_new_header has time complexity O(N) / N per processing element. Under this assumption, we maintain that procedure Evaluate_new_header has average time complexity of O(1) per processing element in each time step. In consequence, each time step of Algorithm 4.9 has time complexity O(1) per processing element.

In each time step, each processing element will generate one position of the N-permutation. There are N! N-permutations. Therefore, each processing element executes N! time steps of Algorithm 4.9. To generate N! N-permutations, the overall running time for each processing element is O(N!).

There are N processing elements. Hence, the total computation cost of Algorithm 4.9 is O(N! * N). This matches the optimal sequential time.

One design consideration is during the cycle-N-perm operation, a new header must be produced in order to maintain the continuity of the algorithm. In this case, one position of the new header is computed in each time step. The new header has size N with the Nth position fixed. Therefore, it can be produced in N-1 time steps. Each cycle-N-perm consists of N-1 time steps. Thus, a new header will be produced in time to start another cycle-N-perm operation and the continuity of the algorithm is maintained.

There are a few drawbacks of Algorithm 4.9. First, the running time of procedure Evaluate_new_header is variable, it may affect the synchronization of the systolic array. When processing element PE_i is executing procedure Evaluate_new_header, its left neighbor PE_{i-1} may have already finished its operations and be waiting for PE_i's output data to start the next time step. A delay here will in turn delay the operations of PE_{i-2}, PE_{i-3}, ..., PE_1 with those processing elements left idle. Therefore, it is preferable to remove this varying time unit within a time step.

The modulo operation restricts the objects being permuted to the integer set {1, 2, 3, ..., N}. Even though some conversion procedure can be apply to maintain the validity of the modulo operation for another set of objects, it will increase the complexity of the overall algorithm.

Finally, the permutations generated are not in any pre-defined order. In the next chapter, we will discuss and design systolic algorithms that generate permutations in lexicographic order.

Chapter 5. Permutation in Lexicographic Order

5. 1 Introduction

The algorithms in chapter four do not generate the permutations in any particular order. Sometimes it is beneficial to generate the permutations in a specific order. Since the problem of generating permutations is inherently exponential, applications are usually interested only in some subsets of the m-permutations. If we can generate the permutations in some pre-defined order, it will be easier for the applications to pick out the subsets they want. Furthermore, some applications require that the permutations generated be sorted. In order to sort the permutations, we need some system to enumerate the permutations.

In this chapter, the ordering system we have chosen is the lexicographic order. It is the most natural ordering system for numbers and alphabets, the data types most frequently encountered in computer science.

First, let us define what lexicographic order means for permutations. Let there be two m-permutations, $P_1 = \{a_1, a_2, ..., a_m\}$ and $P_2 = \{b_1, b_2, ..., b_m\}$. P_1 precedes P_2 in lexicographic order if there exists an i, $1 \le i \le m$, such that $a_j = b_j$ for all $j < i$ and $a_i < b_i$. For example, if N = 3 and the objects are integers from 1 to 3, S = {1, 2, 3}. Then, all 3-permutations of S in lexicographic order are:

(1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1)

Just as in chapter four, we will first look at some sequential algorithms that generate permutations in lexicographic order. Then, we will discuss some existing parallel algorithms. Finally, we will design a new parallel algorithm to efficiently generate permutations in lexicographic order. Without lost of generality, from now on, we will

assume the set of elements being permuted is the set of integers {1,2,3,4, .., N}. Thus, the order of elements can be decided easily by arithmetic comparison. But first, let us look at some criteria regarding algorithms for generating permutations in lexicographic order.

5. 2 Criteria for Permutation Algorithm

Criterion 1. The cost of the algorithm

For a sequential algorithm, the optimal cost for generating and producing as output the m-permutations is $O({}^{N}P_{m} * m)$. This is because there are ${}^{N}P_{m}$ m-permutations, and each one has m elements. An alternative view is to exclude the time to output the permutations, thus the optimal sequential time will be $O({}^{N}P_{m})$.

For a parallel algorithm, the total work (number of processors * running time) must be greater than or equal to the optimal sequential cost. This is because of the law of linear speedup mentioned in Chapter one. A parallel permutation algorithm is said to be cost optimal if its total work is equal to the optimal cost of the sequential algorithm.

There are several parallel algorithms that match the $O({}^{N}P_{m} * m)$ total cost. Designing a parallel algorithm that satisfies the $O({}^{N}P_{m})$ measure is still an open problem. In this thesis, we will try to design a parallel algorithm that satisfies the first measure, a total cost of $O({}^{N}P_{m} * m)$.

Criterion 2. The delay time between consecutive permutations

Since the problem of generating permutations is inherently exponential, the more important issue is the time required by the algorithm between any two consecutive permutations it produces. The delay time is especially important for applications where one output of the computation is the input to another.

Criterion 3. The model of computation

The structures used in the algorithms should be as simple as possible. The simplicity of the model is measured by the following factors:

(1) Size of the local memory.

(2) Modularity, uniformity and regularity.

(3) Operational simplicity.

(4) Communication locality.

A simple and regular computation model can be efficiently implemented in VLSI and, as a result, is more practical regarding hardware cost.

Criterion 4. Space (memory) complexity

How much memory does the algorithm require? For a parallel algorithm, how much memory is required by each processing element. It also affects the simplicity of the computation model as stated in Criterion 3.

Criterion 5. Adaptability

This is important for parallel algorithms. Is the algorithm adaptable in the sense that it can be implemented on different parallel architectures or by using different number of processors? How much is the slow-down when the number of processors is reduced? The adaptability of an algorithm is affected by its computation model.

5. 3 Sequential Algorithms

5. 3. 1 Akl's Sequential Algorithm

The first algorithm was introduced by Akl in [Akl [2]]. It is a natural way to approach the problem of generating permutations in lexicographic order. First, it initializes the current permutation to be the very first permutation in lexicographic order. In our case, the first m-permutation in lexicographic order is (1 2 3 4 ... m). The algorithm then determines if the current permutation is updatable, i.e. , if the permutation is the last one in lexicographic order, which for our case is (N N-1 N-2 ... N-M+2). If it is updatable, it will generate and assign the next m-permutation to be the current permutation. Then, it checks again if it is updatable. Otherwise, the program terminates.

There are two important procedures in this algorithm, namely, Updatable() and Next_permutation(). In order for these procedures to work, we need some data structures to record what the current permutation is and which elements in the integer set {1,2,3,4 ..., N} are in the current permutation. For this, we introduced two array structures - current_perm[p_1, p_2, p_3, ..., p_m] and un_used[u_1, u_2, u_3, ..., u_N]. Array un_used is a boolean array, where u_i is 0 if i is in current_perm, otherwise, it is 1. When the algorithm first starts, current_perm is [1,2,3, ... m] and in array un_used, u_1 to u_m are 0's and the rest are 1's.

First, we will describe procedure Updatable(). The current permutation (p_1 p_2 p_3 ... p_m) is updatable if at least one of its elements p_i can be updated. That means, for p_i, $1 < i < m$, there exists j such that $p_i < j \leq N$ and u_i is equal to 1.

After it has been determined that the current permutation is updatable, we need to compute the next permutation. To do that, first the rightmost p_i and the smallest j which the condition for updatability holds need to be located. Then, p_i is updated to j and u_j has

to be marked as used by setting it to 0. Furthermore, all the elements p_{i+1}, p_{i+2}, ..., p_m to the right of p_i also needed to be updated. This is accomplished as follows: each p_{i+k}, $1 \leq k \leq m-i$, is updated to s if u_s is the kth position in u that is not used, i.e. , it is equal to 1.

With procedures Updatable() and Next_permutation() defined, the permutation algorithm is presented as Algorithm 5. 1.

Algorithm 5. 1 Akl's sequential permutation algorithm

```
Akl_permutation( )
{  current_perm[1. . m] = {1. . m};
   un_used[1. . N] = {1};
   while ( Updatable() )
     Next_permutation();
} /* End procedure Akl_permutation.  */
```

Procedure Next_permutation and Updatable are defined as follows:

```
Next_permutation( )
{ /* mark used elements */
  for (i=1; i≤M; i++)
    un_used[ current_perm[i] ] = 0;
 /* find largest unused element */
 f = N;
 while (!un_used[f]) f--;
 /* find rightmost updatable element */
 k = M+1;  i = 0;
 while ( i == 0 ) {
```

```
    k--; un_used[ current_perm[k] ] = 1;
    if ( current_perm[k] < f )  /* find smallest j such that p_k < j ≤ N and u_j =1 */
    {     j = current_perm[k]+1;
          while ((un_used[j] ) and (j ≤ N))  j++;
          i = k;  current_perm[i] = j;  un_used[ current_perm[i] ] = 1; };
    else f = current_perm[k];
  }; /* end while */
  /* update elements to the right of current_perm[i] */
  for ( k=1; k ≤ M-i; k++ )
  { /* if un_used[s] is kth position in un_used that is 1, current_perm[i+k] = s */
    s = 1; t = 0;
    while ( t != k )
      if ( !un_used[s] ) {  t++;  s++; };
      else s++;
    current_perm[i+k] = s-1;
  }; /* End for */
 for ( k=1; k ≤ N; k++ )  un_used[ current_perm[k] ] = 0;
} /* End procedure Next_permutation.  */


Updatable( )
{ result = TRUE;
  for (i=1; i≤m; i++)
    if (current_perm[i] != N-i-1) { result = FALSE; break; }
  return result;
} /* end procedure Updatable.  */
```

One problem with this algorithm is m cannot be equal to N. This is because to find an un-used element, the algorithm searches for the smallest indexed element in array un_used that is 0. However, in the case of m = N, every element will be used in each permutation. Therefore, the search will always fail. One simple way to overcome this problem is to include an additional element u_0 in array un_used and u_0 is always equal to 0. In this way, an un-used element will always be found. When the algorithm searches for the rightmost updatable element, the smallest indexed element p_j will replace u_0, thus maintaining the correctness of the algorithm.

Another enhancement for this algorithm is to eliminate procedure Updatable(). Instead of testing if the current permutation is updatable every time before Next_permutation() is called, we can simply call Next_permutation( ) ${}^{N}P_{m}$ times. Thus, eliminated an O(m) procedure. One drawback of this is now we have to generate all m-permutations starting from the first one, where in the original algorithm we can generate the ith to jth m-permutations by starting the algorithm with the current permutation set to the ith permutation and call next_permutation() j-i times. However, with a slight modification this may still be accomplished.

Now, we will determine the complexity of procedure Next_permutation(). Each loop in the procedure is independent and takes at most m iterations, therefore, time complexity of procedure Next_permutation is O(m). It means the delay time between successive permutations is O(m). Procedure Next_permutation will be called ${}^{N}P_{m}$ times in the algorithm, as a result, the overall cost of Akl's algorithm is $O({}^{N}P_{m} * m)$. This is in a sense optimal since we are generating ${}^{N}P_{m}$ number of permutation and each permutation is m elements long.

For space complexity, the major data structures in this algorithm are the two arrays current_perm and un_used. Each of them is an array of m elements. Therefore, the

memory requirement of the algorithm is O(m). Since array un_used is a boolean array, memory requirement can be further reduced by declaring un_used to be type unsigned.

5. 3. 2 A Classic Algorithm

The second algorithm is over 180 years old. It was first published by Fischer and Krause in 1812. Over the years, this algorithm has been rediscovered many times. Dijkstra [Dijkstra [7]] cited the algorithm as an example to illustrate a "dramatic improvement in the state of the art" of computer programming since it can be easily expressed in modern computer languages.

Like the other algorithms being discussed in this chapter, this algorithm is based on the idea of producing each permutation from its lexicographic predecessor. This algorithm has been discussed in the Sedgewick's paper. In his paper, Sedgewick only presented the algorithm to generate permutations in reverse lexicographic order (i.e. reading the lexicographic sequence backward and the permutation from right to left, for example, all the 3-permutations in reverse lexicographic order are 123, 213, 132, 312, 231, 321). However, with a small change to Sedgewick's algorithm by reversing the direction of comparisons, his algorithm can be used to generate permutations in lexicographic order.

This algorithm only works for the case where m = N. It starts with the first permutation of N elements in lexicographic order, i.e. P[1 ... N] = (1 2 3 4 5 ... N). To compute each successive permutation, we scan the permutation from right to left searching for the first element P[i] < P[i+1]. Then, this element P[i] is swapped with the smallest element to its right that is larger than P[i]. Finally, all the elements to the right of P[i] are reversed, i.e. P[i+1] ... P[N] will become P[N] ... P[i+1] .

For example, to find the next permutation from (2 1 3 6 8 7 5 4), we first scan the permutation from right to left and find 6 to be the first element smaller than its right

neighbor. Then, 6 is swapped with the smallest element to its right that is larger than 6, in this case 7. Now, the permutation became (2 1 3 7 8 6 5 4). Finally, the elements to the right of 7 are reversed and the permutation became (2 1 3 7 4 5 6 8). Therefore, the permutation following (2 1 3 6 8 7 5 4) is (2 1 3 7 4 5 6 8).

The algorithm is presented as Algorithm 5.2. It uses the notation of Algorithm 5.1. It computes the next permutation from the current permutation. In fact, it can be used to replace procedure Next_permutation() in Algorithm 5. 1.

Algorithm 5. 2 A classic sequential permutation algorithm

```
Classic_next_permutation()
{ i = M;
 found = FALSE;
 while (!found)
 { if ( current_perm[i] > current_perm[i-1] )
     found = TRUE;
   else
     i--; };
 i--;
 /* i have the first element to exchange */
 k = N+1;
 for ( j = i+1; j ≤ M; j++)
  if ((current_perm[i] < current_perm[j]) && (current_perm[j] < k))
  { k = current_perm[j];
     e = j; };
```

```
/* e have the next bigger element than i, exchange i and e */
 swap(current_perm[i], current_perm[e]);
  /* reverse current_perm[i+1 - M] */
 for (count = i+1; count ≤ M; count++)
  temp_perm[count] = current_perm[count];
 temp = M;
 for (count = i+1; count ≤ M; count++)
 { current_perm[count] = temp_perm[temp];
  temp--;
 };
} /* End procedure Classic_next_permutation */
```

Procedure Classic_next_permutation() has running time O(m), the same complexity to Akl's algorithm. However, it has a smaller constant which means it should be faster than Akl's algorithm. Concerning space, this algorithm does not need to keep track of which elements are used. Therefore, it also has a smaller memory requirement.

The biggest problem with this algorithm is it will only work for the case where m = N. In order to modify this algorithm to work for m ≠ N, it will need to keep track of elements that are not in the current permutation. It means it will probably eliminate its time and space advantages over Akl's algorithm. However, the idea of this algorithm is still much simpler and can be implemented easily in most of the currently popular computer languages. This is probably the reason why this algorithm has been rediscovered so many times in its history of almost two centuries.

5. 4 Parallel Algorithms

5. 4. 1 Parallel Version of Algorithm 5.1

The first parallel algorithm we are going to discuss here is a direct parallel version of Algorithm 5.1. This algorithm was also introduced by Akl [Akl [2]]. The idea is to parallelize each sequential part of the procedure Next_permutation in Algorithm 5.1 using m processors. Recall that Algorithm 5.1 consists of mainly three steps:

1. Assume the current permutation is (P_1 P_2 P_3 ... P_m), the algorithm first determines if the current permutation is updatable. If the current permutation is not updatable, the algorithm is terminated.
2. If the current permutation is updatable, the algorithm checks if element P_m can be incremented. If it can be incremented, the algorithm increments it and terminates.
3. If element P_m cannot be incremented, the algorithm finds the first element to the left of P_m that is smaller than its right neighbor. Then, the algorithm increments this element and updates all the other elements to its right.

The parallel version of procedure Next_permutation uses five parallel procedures - Broadcast, Allsums, Minimum, Maximum and Scan - to implement those three steps. They are described as follows:

1. Procedure Broadcast(a, m, x) distributes the value a among processing elements PE_1, PE_2, ... , PE_m by using array x_1, x_2, ... , x_m in the shared memory.

```
Broadcast(a, m, x)
{  int local_mem_i; /* local memory at processor P_i */
   for processor P_1
     read(a, local_mem_1);   /* read value a into local memory local_mem_1 */
     write(local_mem_1, x_1); /* write value in local_mem_1 into x1 */
   for ( i = 0; i ≤ (log m - 1); i++)
    for ( j = 2^i + 1; j ≤ 2^(i+1); j++)
      for each processing element PE_j do in parallel
        read(x_(j-2^i), local_mem_i);
        writes(local_mem_i, x_j);
} /* End procedure Broadcast.  */
```

Procedure broadcast takes O(log m) time and has space requirement O(m).

2. Procedure Allsums(x_1,x_2, ... , x_m) computes the prefix sum of array x_1, x_2, ..., x_m. At the end of the procedure, $x_i = x_1 + x_2 + ... + x_i$ for $1 \le i \le m$.

```
Allsums(x_1, x_2, ..., x_m)
{  for ( j = 0; j ≤ log m - 1; j++)
    for ( i = 2^j + 1; j ≤ N; j++)
      for each processing element PE_i do in parallel
        obtains x_(i-2^j) from P_(i-2^j)
        x_i = x_(i-2^j) + x_i
} /* End procedure Allsums.  */
```

Procedure Allsums takes O(log m) time and has space requirement O(m).

3. Procedure Minimum(x_1, x_2, ..., x_m) finds the smallest element in array x_1, x_2, ..., x_m and returns it in x_1.

```
Minimum(x1, x2, . . . . . , xm)
{ for ( j = 0; j ≤ log m - 1; j++ )
    for ( i = 1; i ≤ m; i += 2^(j+1) )
      for each processing element PEi do in parallel
        Pi obtains x(i+2^j)
        if ( x(i+2^j) < xi )
          xi = x(i+2^j);
} /* End procedure Minimum. */
```

Procedure Minimum takes O(log m) time and has space requirement O(m).

4. Procedure Maximum(x_1, x_2, ..., x_m) finds the largest element in array x_1, x_2, ..., x_m and returns it in x_1.

```
Maximum(x1, x2, ..., xm)
{ for ( j = 0; j ≤ log m - 1; j++ )
    for ( i = 1; i ≤ m; i += 2^(j+1) )
      for each processing element PEi do in parallel
        Pi obtains x(i+2^j)
        if ( x(i+2^j) > xi )
          xi = x(i+2^j);
} /* End procedure Maximum. */
```

Procedure maximum takes O(log m) time and has space requirement O(m).

5. Procedure Scan(p_s, N, x_1 ... x_m, un_used) is being used to help searching for the next available integer to increment element p_s. It determines which of the m integers $p_s + i$, $1 \leq i \leq m$, satisfies the following conditions:

(i) It is smaller than or equal to N.

(ii) It is not in the current permutation, i.e. , un_used[p_s+i] = 1.

If $p_s + i$ satisfies the above conditions, x_i is assigned $p_s + i$. Otherwise, $x_i = \infty$.

```
Scan(p_s, N, x_1 ... x_m, un_used)
{  for ( i = 1; i ≤ m; i++ )
     for each processing element  PE_i do in parallel
       if (p_s + i ≤ N) and (un_used[p_s+1])
         x_i = p_s + i;
       else
         x_i = ∞;
} /* End procedure Scan.  */
```

Procedure scan takes O(1) time and have space requirement O(m).

With the five parallel procedures defined, we can now present the parallel version of Algorithm 4.1 as Algorithm 5.1. Algorithm 5.1 uses the same data structures as Algorithm 4.1, including the arrays current_perm and un_used.

Algorithm 5. 3 Parallel version of Algorithm 5. 1

```
/* Produce all m-permutation of N objects */
Parallel_Akl_permutation()
{  /* Produce the first permutation */
   for ( i = 1; i ≤ m; i++ )
     for each processing element PE_i do in parallel
       current_perm[i] = i;
       output( current_perm[i] );
   /* Initialize array un_used */
   for ( i = 1; i ≤ ⌈N / m⌉; i++ )
     for ( j = 1; j ≤ m; j++ )
       for each processing element PE_j do in parallel
         k = (i - 1) * m + j
         if ( k ≤ N )
           un_used[k] = 0;
   for ( t = 1; t ≤ NP_m; t++ )
     Parallel_next_permutation();
   /* Clean up and output the current permutation.  */
   for ( i = 1; i ≤ k; i++ )
     for each processing element PE_i do in parallel
       un_used[ current_perm[i] ] = 1;
       output[ current_perm[i] ];
} /* End procedure Parallel_Akl_permutation.  */
```

Procedure Parallel_next_permutation is defined as follows.

```
Parallel_next_permutation()
{ /* Step 1 : Initialize un_used */
  for ( i = 1; i ≤ m; i++)
    for (each processing element PE_i do in parallel) un_used[ current_perm[i] ] = 0;
  /* Step two: Check if the current_perm[m] can be incremented.  If so,
     search for available element to increment it with.  */
  Broadcast(current_perm[m], m, x);
  Scan(current_perm[m], N);
 /* If more than one element can be used to increment the right most element, find the
     smallest one.  */
  Minimum(x);
  if ( x[1] != ∞ ) {
    un_used[ current_perm[m] ] = 1;
    current_perm[m] = x[1];
    k = m - 1; };
  else
  /* Step three : current_perm[m] cannot be increment, find the right most
     element to its left that can be increment, current_perm[k]*/
   for ( i = 1; i ≤ m - 1; i++)
    for each processing element PE_i do in parallel
      if (current_perm[i] < current_perm[i+1])  x[i] = i;
      else x[i] = -1;
   Maximum(x);  k = x[1];
   Broadcast(k, m, x);
   Broadcast(current_perm[k], m, x);
```

```
   /* Step four : Increment current_perm[k] */
   for ( i = k; i ≤ m; i++ )
     for each processing element PE_i do in parallel
       un_used[ current_perm[i] ] = 1;
   Scan(current_perm[k], N);
   Minimum(x);
   current_perm[k] = x[1];
   un_used[ current_perm[k] ] = 0;
   /* Step five : update element to the right of current_perm[k] */
   for ( i = 1; i ≤ m; i++)
    for each processing element PE_i do in parallel
       x[i] = un_used[i];
   Allsums(x);
   for (i = 1; i ≤ m; i++)
    for each processing element PE_i do in parallel
      if ( x[i] ≤ ( m - k) ) and ( un_used[i] )
        current_perm[ k+x[i] ] = i;
}  /* End procedure Parallel_next_permutation */
```

Now, we will show how Procedure Parallel_next_permutation operates to update a permutation.

Assume that we are generating 4-permutations from 5 objects, and the current permutation is (2 3 4 5). It means m = 4, and four processors will be used.

Step one : array un_used is set to be [1, 0, 0, 0, 0], indicating that only object 1 is not used.

Step two : Element current_perm[4] is broadcasted to the four processors. Procedure Scan is then called to determine if current_perm[4] is updatable. After procedure Scan returns, array x becomes [∞, ∞, ∞, ∞]. It shows that current_perm[4] is not updatable.

Step three : Since current_perm[4] is not updatable, the procedure examines the elements to its left to search for an element current_perm[k] that current_perm[k] < current_perm[k+1]. After the search, array x becomes [1, 2, 3, -1]. By calling procedure Maximum, k is found to be 3, and current_perm[3] is the rightmost element we can increment. 3 and current_perm[3] are broadcasted to all the processors.

Step four : To increment current_perm[3], first array un_used is set to be [1, 0, 0, 1, 1]. Now, procedure Scan is called and array x becomes [5, ∞, ∞, ∞]. Next, procedure Minimum returns 5. It means we should increment current_perm[3] to 5.

Step five : Elements to the right of current_perm[3] are updated. We do this by first setting array x to be the first four elements of array un_used, i.e. x = [1, 0, 0, 1]. After procedure Allsums is called, x = [1, 1, 1, 2]. Since only x[1] is smaller than or equal to (m - k) and not used, current_perm[4], the only element to the right of current_perm[3], is updated to 1. As a result, this yields the next permutation (2 3 5 1).

Now, we will analyze the complexity of Algorithm 5.3. The major time consumptions of the algorithm are the calls to Parallel_next_permutation. Each step of procedure Parallel_next_permutation takes at most O(log m) time. As a result, Parallel_next_permutation has time complexity O(log m). It is called ${}^{N}P_{m}$ times. Therefore, the overall running time of Algorithm 5.3 is $O({}^{N}P_{m} * \log m)$.

Since m processors are used, the total cost of Algorithm 5.3 is $O({}^{N}P_{m} * \log m * m)$. Since an optimal sequential algorithm has time complexity $O({}^{N}P_{m} * m)$, Algorithm 5.3 is not optimal.

The main data structures of the algorithm are arrays x, current_perm and un_used. They have space complexity of O(m), O(m) and O(N) respectively.

This algorithm is not optimal and is not adaptive. It requires the use of m processors to generate all m-permutations. However, it does has a low delay time between successive permutation - O(log m) comparing to O(m) of sequential algorithms.

Moreover, each access to array x, current_perm and un_used is independent regarding each processing element. There is never an instance where more than one processor is trying to access the same memory location. Therefore, this algorithm can be implemented on the EREW shared memory model. This weakest model of parallel computation allows the algorithm to be implemented easily on any parallel system.

One more observation about this algorithm is the parallelization in the steps of producing each permutation. This reduces the delay time between each successive permutations. However, the permutations are still being generated one by one. Thus, the running time will not be lower than $O({}^{N}P_{m})$. Given multiple processing elements, it is possible to generate more than one permutation at the same time. In the following sections, we will look at algorithms that generate more than one permutation in each instance. We will also examine how this will affect all aspects of the algorithm.

5. 4. 2 Ranking and Unranking of Permutations

Before we get into the next algorithm, we will discuss the ranking and unranking of permutations on which the next algorithm will be based.

The permutations in this chapter are being generated in lexicographic order. Therefore, each permutation has its precise order to be generated. In other words, it is valid to call a permutation the ith permutation to be generated in this order, and the value i is distinct. In our case of generating m-permutations of N objects, there is an one-to-one correspondence between the value i and each m-permutation, where $1 \le i \le {}^{N}P_{m}$. The next question is, given the value i, is it possible to compute the permutation corresponding to this value? Looking at this from another direction, given a specific permutation, is it possible to compute the corresponding value? This introduces two functions, namely, the ranking of a permutation and the unranking of a value to produce the corresponding permutation. We will define the two functions as follows:

(1) Let P_k to be one of the m-permutations of N objects, $\text{rank}(P_k) = D$, where $1 \le D \le {}^{N}P_{m}$. If P_j is another m-permutation and $P_j < P_k$, then, $\text{rank}(P_j) < \text{rank}(P_k)$. Furthermore, $\text{rank}(P_j) = \text{rank}(P_k)$, if and only if P_j is equal to P_k.

(2) Let P_k be one of the m-permutations of N objects and $D = \text{rank}(P_k)$, then, $\text{unrank}(D) = \text{rank}^{-1}(D) = P_k$. Note that $\text{rank}^{-1}(D)$ is the inverse of rank(D).

Now, we will first describe the function rank(P). Let $P = (p_1\ p_2\ p_3 \ldots p_m)$, we define the sequence $R = (r_1, r_2, r_3, \ldots, r_m)$, where,

$$r_i = p_i - i + \sum_{j=1}^{i-1} (p_i \Leftarrow p_j), \text{ where } p_i \Leftarrow p_j = 1 \text{ if } p_i < p_j$$

$$= 0 \text{ if otherwise.}$$

Then,

$$rank(P) = 1 + \sum_{i=1}^{m-1} \left(r_i * \prod_{j=0}^{m-i-1} (N-i-j) \right).$$

Let us try it on an example. Assume P = (1 5 2) is one of the 3-permutations of 5 objects. We want to compute rank(P). First, we will compute the sequence R = (r1, r2, r3).

$$r_1 = p_1 - 1 + \sum_{j=1}^{0} (p_1 \Leftarrow p_j) = 1 - 1 + 0 = 0.$$

$$r_2 = p2 - 2 + \sum_{j=1}^{1} (p2 \Leftarrow p_j) = 5 - 2 + 0 = 3.$$

$$r_3 = p_3 - 3 + \sum_{j=1}^{2} (p_3 \Leftarrow p_j) = 2 - 3 + 1 = 0$$

Therefore, R = (0, 3, 0). Now,

$$\text{rank}(1\ 5\ 2) = 1 + \sum_{i=1}^{2} \left(r_i * \prod_{j=0}^{2-i} (N-i-j) \right)$$

$$= 1 + 0 * (5-1-0) * (5-1-1) + 3 * (5-2-0) = 10$$

Therefore, (1 5 2) is the 10th 3-permutation of 5 objects, which is exactly correct.

Next, we will describe unrank(D) = P = (p_1 p_2 p_3 ... p_m), which is the inverse of rank(P). Let D = rank(P). First, we will define the sequence R = (r_1, r_2, r_3, ..., r_m), where,

$$r_i = \left\lfloor \left(D - 1 - \sum_{j=1}^{i-1} r_j * \prod_{k=0}^{m-j-1} (N-j-k) \right) / \prod_{k=0}^{m-i-1} (N-i-k) \right\rfloor$$

Then, unrank(D) = P = (p_1 p_2 p_3 ... p_m), where

$$p_i = r_i + i - d_i, \text{ for } 1 \le i \le m$$

and d_i is the smallest non-negative integer such that

$$d_i = \sum_{j=1}^{i-1} ((r_i + i - d_i) \Leftarrow p_j)).$$

Now, we will try that on the previous example and compute unrank(10) for 3-permutations of 5 objects. First, we will compute the sequence R = (r_1 r_2 r_3).

$$r_1 = \left\lfloor \left(10 - 1 - \sum_{j=1}^{0} r_1 * \prod_{k=0}^{2-j} (5-j-k) \right) / \prod_{k=0}^{1} (5-1-k) \right\rfloor = 0$$

and similarly, $r_2 = 3$ and $r_3 = 0$.

Therefore, unrank(10) = P = (p_1 p_2 p_3), where

$p_1 = r_1 + 1 - d_1$,

and d_1 is the smallest non-negative integer such that

$$d_1 = \sum_{j=1}^{i-1} ((0 + 1 - d_1) \Leftarrow p_j)).$$

It turns out that $d_1 = 0$. With similar computation, d2 = 0 and d3 = 1.

Finally, p1 = 0 + 1 - 0 = 1,

p2 = 3 + 2 - 0 = 5,

p3 = 0 + 3 - 1 = 2.

Therefore, P = (1 5 2) which is the correct answer.

In the algorithm to generate permutations, we are only interested in the unrank(D) function. Moreover, in the case where m = N, there is a simpler way to compute the unrank function without constructing the sequence R.

Assuming we are trying to determine the kth N-permutations in lexicographic order, first we write

$$k = 1 + \sum_{i=1}^{n-1} l_i * (N - i)! \quad \text{where } l_i \leq N - i.$$

This representation is unique for any k and N.

Then, the kth N-permutations is (p_1 p_2 p_3 ... p_N), where

$p_1 = l_i + 1$, *and*

p_j, $2 < j < N-1$, *is the smallest integer in* $\{l_j+1, ..., N\} - \{p_1, p_2, ..., p_{j-1}\}$ *such that*

$$p_j = 1 + l_j + \sum_{k=1}^{j-1} p_k \Leftarrow p_j$$

Again we will try this on an example. Assume that we are computing the 10th 5-permutations of 5 objects. First, we write

$$10 = 1 + l_1 * 4! + l_2 * 3! + l_3 * 2! + l_4 * 1!$$
$$= 1 + 0 * 4! + 1 * 3! + 1 * 2! + 1 * 1!$$

Therefore, $l_1 = 0$, $l_2 = 1$, $l_3 = 1$, $l_4 = 1$. Then, the 10th 5-permutation of five objects, (p1 p2 p3 p4 p5) is

$p_1 = 1$

p_2 = the smallest integer in {3, 4, 5} - {1} = {3, 4, 5} such that

$$p_2 = 1 + 1 + \sum_{k=1}^{j-1} p_k \Leftarrow p_j,$$ and in this case $p_2 = 3$.

With similar computation, p3 = 4, p4 = 5 and p5 = 2

Therefore, the 10th 5-permutations of 5 objects is (1 3 4 5 2).

There are many ways to implement the unrank function. The following algorithm for the unrank function is given by Akl [Akl[2]].

```
/* Compute the Dth m-permutations of N objects as P[1. . m]. */
Unrank(N, m, D, P)
 int r[N], i, j, k, a, b;
{ D--;
  for (i = 1; i ≤ N; i++)  r[i] = 0;
   a = 1;
 for (i = m - 1; i ≥ 1; i--)  j = j * ( N - m + i );
 for ( i = 1; i ≤ m; i++ ) {
   b = ⌊ D / a ⌋ ;
   D = D - ( a * b);
   if ( N > i ) a = a / (N - i);
   k = 0;
   j = 0;
   /* Find the ( b + 1)-th position in s that is equal to 0 */
   while ( k < (b + 1) ) {
     j++;
     if ( r_j = 0 )
       k ++; };
   P[i] = j;
   r[j] = 1
 } /* End for */
} /* End procedure Unrank */
```

The main cost of this algorithm is the last for loop, it takes O(m*N) time. Therefore, the cost to compute the Dth m-permutation of N objects is O(m*N).

With the unrank function defined, we can devise an algorithm to generate all m-permutations of N objects. The first idea which comes to mind is to replace procedure Next_permutation with procedure Unrank. Therefore, an algorithm that generates permutation will be:

```
Permutation()
{ for (i = 1; i ≤ NPm; i++)
    /* compute the ith permutation */
    unrank(N, m, i, P);
    Output(P); }.
```

However, this algorithm is very inefficient. In fact, it has running time of $O({}^{N}P_{m} * m * N)$ compared with the optimal time of $O({}^{N}P_{m} * m)$. Nevertheless, we can use the function Unrank to devise an optimal parallel algorithm. The idea is to partition the permutations into independent sets. Each processing element will be assigned a separate set, and each processing element is responsible for producing the permutations of its corresponding set.

Assuming we are using K processing elements, we will use the Unrank function to partition the permutations into K sets. In other words, each processing element P_i, $1 \le i \le K$, will produce the permutation set of $\lceil N / K \rceil$ m-permutations starting with the $((i - 1) * \lceil N / K \rceil + 1)$-th m-permutation. The algorithm is presented as Algorithm 5.4.

Algorithm 5.4 Permutation algorithm using the Unrank function

```
Permutation_with_unrank(N, m)
{  for ( i = 1; i ≤ K; i++ )
     for each processing element PE_i do in parallel {
       D = (i - 1) * ⌈N / K⌉ + 1;
       unrank(N, m, D, current_perm);
       for ( j = 1; j ≤ NP_m - 1; j++ ) {
         next_permutation;
         output(current_perm); }; };
} /* End procedure Permutation_with_unrank */
```

Each processing element operates in an independent fashion. The running time for each processor is $O(m*N) + O(\lceil {}^{N}P_m / K \rceil * m)$. If N is smaller than $\lceil {}^{N}P_m / K \rceil$, then the running time for each processor is $O(\lceil {}^{N}P_m / K \rceil * m)$. There are K processing elements, therefore, the total cost of the algorithm is $K * O(\lceil {}^{N}P_m / K \rceil * m) = O({}^{N}P_m * m)$ which matches the optimal sequential time. It means that the algorithm is optimal if we use less than $\lceil {}^{N}P_m / N \rceil$ number of processing elements.

Furthermore, if we restrict the number of processing elements being used, we can eliminate the use of the Unrank function. To generate all the m-permutations of N objects, we will use N processing elements. The way we partition the permutation sets is each processing element PE_i, $1 \le i \le N$, will generate all the permutations with leading element i. The only problem is that we need to compute the initial current_elem for each processor. However, it is not difficult to see that the first permutation for each processor P_i is (i and (the smallest m - 1 elements except i)). Algorithm 5.5 is essentially the same as Algorithm 5. 4 except it does not use the unrank function.

Algorithm 5. 5 Permutation of all N-permutation without using Unrank

```
Permutation_without_unrank(N, m)
{  for (i = 1; i ≤ N; i++)
    for each processing element PE_i do in parallel {
     current_perm[1] = i;
     for ( j = 1; j ≤ m + 1; j++ ) {
      k = 2;
      if ( j != i ) {
       current_perm[k] = j;
       k ++;
      } /* End if */  };  /* End for */
     for ( j = 1; j ≤ (N-1)! / (N-m)! - 1; j++ ) {
      output(current_perm);
      next_permutation; };  } /* End for */
} /* End procedure Permutation_without_unrank.  */
```

The running time of each processing element is O([(N-1)! / (N-m)!] * m). There are N processing elements, therefore, the total cost of the algorithm is O(N * [[(N-1)! / (N-m)!] * m]) which is equal to O([N! / (N-m)!] * m) = $O({}^{N}P_{m} * m)$. Therefore, this parallel algorithm is optimal.

There are some points that are worth noting regarding these approaches. First, there is no communication between the processing elements. Once each processing element gets assigned its processor number, we don't even need a shared memory. Therefore, this algorithm can be implemented on a EREW model and can be easily implemented with any parallel system.

Furthermore, the first approach using the Unrank function is more adaptive. The second approach needs N processing elements to generate all the N-permutations for N objects. In many case, it is more practical to assume the number of processors in a parallel system is not only fixed but also smaller than the size of the problems.

Another point which needs to be made is if ${}^{N}P_{m}$ is not a multiple of K, some of the processors will be left idle at the last time step. Therefore, it may be beneficial to choose a K that can evenly divide ${}^{N}P_{m}$.

A problem with this algorithm is whether we can really call the permutations generated by this approach to be in lexicographic order. Each processing element does generate its own set of permutations in lexicographic order. However, at the same instance, permutations generated by the K processing elements are not in lexicographic order. For example, in the first iteration of Algorithm 5.4, processing element PE_1 produced permutation (1 2 3 ... N) while PE_2 produced (2 1 3 ... N). They are clearly not in lexicographic order. In other words, the Nth permutation in lexicographic order will not necessarily be the Nth permutation generated by the algorithm. Therefore, it is not easy to generate a subset of the N-permutations with this approach.

Finally, there is no reason why we cannot use a parallel algorithm to compute the Next_permutation. In this case, we will have a grid connected architecture (Figure 5.1). Of course, that increases the complexity of the algorithm. The total cost of the algorithm becomes $O({}^{N}P_{m}$ * m * running time of parallel_next_permutation). In order for the algorithm to match the optimal sequential time, the parallel Next_permutation procedure must be running in constant time. This is not easy. A parallel next_permutation procedure such as Algorithm 5.3, which is O(log m), will put the algorithm over the optimal cost.

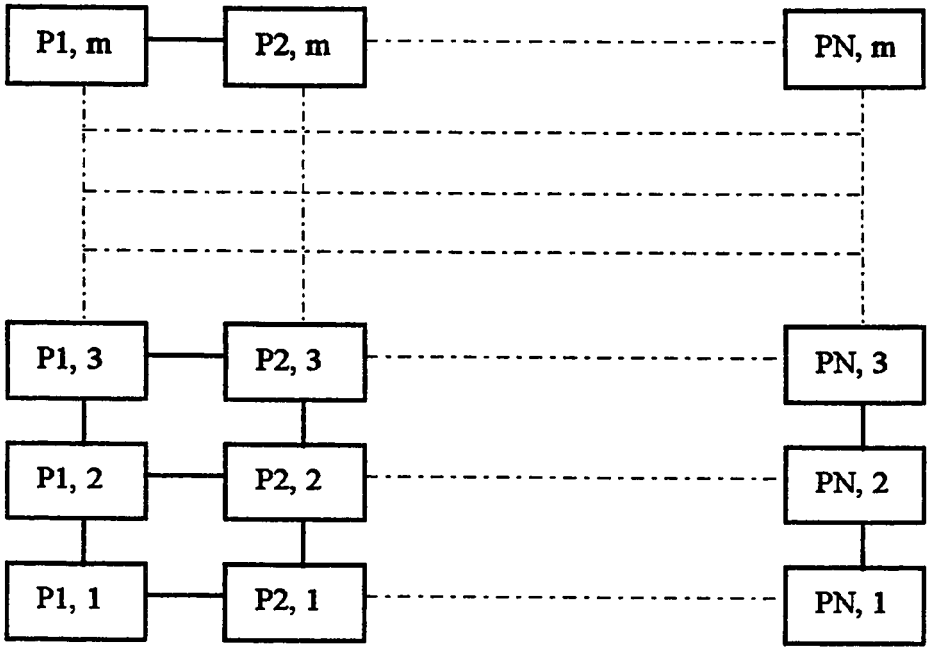


Figure 5.1 : Grid connected processing elements.

However, the two dimensional approach does reduce the delay time between successive permutations to O(log m), which in other cases are O(m). In algorithms such as Algorithm 5.4 and 5.5, the parallelism is focused on the whole set of permutations. The generation of each permutation basically remains sequential. On the other hand, in algorithms such as Algorithm 5.3 and the systolic algorithms which will be discussed in the following sections, the parallelism is focused on generating each permutation. For applications with more concern on the delay time for each permutation, the later may be a more practical approach.

5. 4. 3 Systolic Algorithm

In section 4. 3. 4, we have presented a systolic algorithm for generating all the N-permutations. However, that algorithm does not generate the permutations in any order. In this section, we will first present a systolic algorithm introduced by Akl and Meuer [Akl & Meuer [3]]. It generates all the N-permutations in lexicographic order with optimal cost and constant delay time. Then, we will design a systolic algorithm with similar complexity.

The systolic array we are using here is essentially the same architecture which was used in section 4. 3. 4. Recall that it is a linear array of processing elements which in each time step perform the following:

(1) Read the data from its input links.

(2) Execute one iteration of the systolic algorithm.

(3) Send the data to its output links.

In this section, we will use the same notations such as input links, output links and time steps, etc. We will also assume the N objects being permuted are the integers {1, 2, 3, ..., N}. However, this will not affect the generality of the algorithms. Unlike Algorithm 4.9, the algorithm presented in this section does not perform any arithmetic on the objects. Therefore, the objects can essentially be anything. Now, we will first present Akl's algorithm.

Systolic Algorithm by Akl and Meuer

Their algorithm generates all the N-permutations with N processing elements. Each processing element will be responsible for one position of each permutation. They are indexed from right to left by integers 1 to N. Each processing element PE_i maintains a variable D.i storing the value of the corresponding position of the permutation. At the beginning of the algorithm, each variable D.i is initialized to N - i + 1. Thus, resulting in

the initial state shown in Figure 5.2. There are other variables being used in this algorithm which will be explained later.

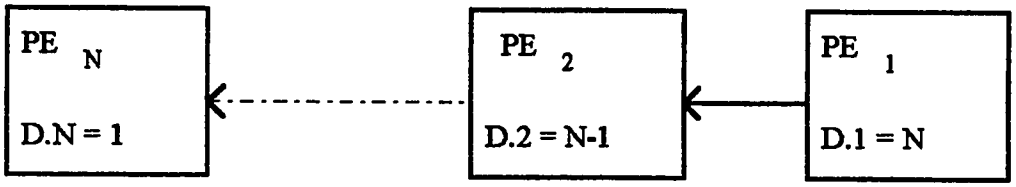


Figure 5.2: Initial state of the systolic array.

For Akl's algorithm, we will choose an integer s, where s is the smallest positive integer such that $(s - 1)! \geq 2N$. For example, if $N = 9$, then $s = 5$. For each processing element PE_i, if $i < s$, we will call it the "small numbered" (SN) processing element. If $i \geq s$, we will call it the "large numbered" (LN) processing element. The reason we defined s this way will be clear when we start presenting the algorithm. To explain the algorithm better, we list the first 25 permutations of 9 objects in Figure 5.3.

As you can see from Figure 5.3, processing elements with higher indices do not change their values in D.i as often as processing elements with lower indices. In other words, LN processing elements do not change their values in D.i as often as SN processing elements. In fact, each processing element PE_i will produce the same value in D.i $(i-1)!$ times.

To explain the algorithm, we introduce two notations, j-block and j-run. In a j-block, processing elements PE_i, $i \geq j$, do not change their values in D.i. In Figure 5.3, permutations (1) to (6), (7) to (12), (13) to (18) and (19) to (24) are all 4-blocks. A j-run is a j+1 block. In other words, only the first j processing elements change their values in D.i. In Figure 5.3, permutations (1) - (24) is a 5-run. Moreover, a j-block contains $(j-1)!$ permutations while a j-run contains $j!$ permutations.

	PE_9	PE_8	PE_7	PE_6	PE_5	PE_4	PE_3	PE_2	PE_1
1)	1	2	3	4	5	6	7	8	9
2)	1	2	3	4	5	6	7	9	8
3)	1	2	3	4	5	6	8	7	9
4)	1	2	3	4	5	6	8	9	7
5)	1	2	3	4	5	6	9	7	8
6)	1	2	3	4	5	6	9	8	7
7)	1	2	3	4	5	7	6	8	9
8)	1	2	3	4	5	7	6	9	8
9)	1	2	3	4	5	7	8	6	9
10)	1	2	3	4	5	7	8	9	6
11)	1	2	3	4	5	7	9	6	8
12)	1	2	3	4	5	7	9	8	6
13)	1	2	3	4	5	8	6	7	9
14)	1	2	3	4	5	8	6	9	7
15)	1	2	3	4	5	8	7	6	9
16)	1	2	3	4	5	8	7	9	6
17)	1	2	3	4	5	8	9	6	7
18)	1	2	3	4	5	8	9	7	6
19)	1	2	3	4	5	9	6	7	8
20)	1	2	3	4	5	9	6	8	7
21)	1	2	3	4	5	9	7	6	8
22)	1	2	3	4	5	9	7	8	6
23)	1	2	3	4	5	9	8	6	7
24)	1	2	3	4	5	9	8	7	6
25)	1	2	3	4	6	5	7	8	9

Figure 5. 3: PE_i and D.i for the first 25 permutations.

The basic idea of this algorithm is the LN processing elements will always produce the same values in D.i until they are told to change. The SN processing elements will be responsible for producing the permutations with objects D.i, $1 \leq i < s$. In order to do that, we need two algorithms. Given the first permutation of an s-block, (let's call it the header of an s-block), we need an algorithm that will compute the header of the next s-block. For our example in Figure 5.3, s = 5, each 5-block contains 25 permutations. The header of the first 5-block is (1 2 3 4 5 6 7 8 9). The header of the next 5-block is (1 2 3 4 6 5 7 8

9). Therefore, we need an algorithm that will compute (1 2 3 4 6 5 7 8 9) from (1 2 3 4 5 6 7 8 9). Moreover, it needs to accomplish that within 25 steps so that the new header will be ready for the generation of the next 5-block..

On the other hand, we need an algorithm that will generate all the permutations within an s-block. The time requirement of this algorithm will control the delay time between successive permutations. We will first present the algorithm for computing the header of the next s-block. Then, we will present the algorithm for generating permutations within an s-block.

Computing the header of the next s-block

As each s-block begins, it can also be the beginning of a k-run, where $k \geq s$, $D.k < D.k-1$. In our example, $k = s$. However, assuming (4 3 9 8 7 1 2 5 6) is a header for a 5-block, the next header of a 5-block (4 5 1 2 3 6 7 8 9) is also a header for a 6-block, 7-block and 8-block. We will call this k the ***turning point***. The algorithm will need to determine this turning point, increment it and update the processing element PE_i, $i < k$. The algorithm is as follows:

Given a header of the s-block, we will first form a "train" with processing elements PE_{s-1}, PE_{s-2}, PE_{s-3}, ..., PE_1. Each processing element will send its "D value" to its left neighbor along with the train. Therefore, initially the train consists of D.s-1, D.s-2, D.s-3, ..., D.1. As the head of the train passes PE_i, $i \geq s$, the D values D.s, D.s+1, ..., D.k are appended to the tail of the train. The train will keep moving left until the head of the train reaches a processing element PE_k, where $k \geq s$ and $D.k < D.k-1$. And PE_k will be the turning point. At this point, the train starts to turn around and moves back toward PE_1. As the train goes pass the turning point, the first D value smaller than D.k is exchanged with it. Then, the train just keeps moving toward PE_1 while the D values are being

dropped off one by one at each processing element. After the last D value is dropped off at PE_1, we will have the next header of the s-block. We will try this on an example with permutation (1 2 3 4 5 6 7 8 9). Figure 5.4 demonstrates the movement of the train.

	PE_9	PE_8	PE_7	PE_6	PE_5	PE_4	PE_3	PE_2	PE_1
D. i	1	2	3	4	5	6	7	8	9
(1)					←	6	7	8	9
(2)					6	7	8	9	
(3)		9	8	7	5	→			
(4)			9	8	7	5	→		
(5)					9	8	7	→	
(6)							9	8	→
(7)									9
(8)	1	2	3	4	6	5	7	8	9

Figure 5. 4: Computing the header of the next 5-block from (1 2 3 4 5 6 7 8 9).

Initially, the train consisted of 6789 at step (1). It kept moving to the left until the head of the train - 6 - reached PE_s, s = 5, at step (2). At this point, the train turned around and moved back toward PE_1 at step (3). At the same time, D.5 is exchanged with 6 and 5 became the head of the train. Now, the train consisted of 5789 is moving back toward PE_1. When the head of the train reached PE_4, 5 is dropped off and 7 became the head of the train at step (4). When a value is dropped off at PE_i, it was assigned to D.i. At step (5), 7 is dropped off at PE_3. At step (6), 8 is dropped off at PE_2. Finally, 9 is dropped off at PE_1 and we have the next header (1 2 3 4 6 5 7 8 9).

In the worse case, the train will travel back and forth through all N processing elements. Therefore, it will take at most 2N steps to compute the new header. As we mentioned earlier, the header must be produced within an s-block and each s-block consists of (s - 1)! permutations. Therefore, the header will be produced in time if $s \geq 2N$. Hence, that is the reason we defined s the way we did on page 130.

Another point is what if we cannot find a turning point as the train moves toward PE_N. The only time we cannot find a turning point is when the given permutation is (N . . . . . 3 2 1); this is also the last permutation in lexicographic order and the termination point. Therefore, if a turning point is not find, the processing elements should recognize the end of the algorithm and terminate.

On the other hand, we can see the new header may be ready even before the current s-block has ended. Therefore, the LN processing elements must keep outputting the old header until the new s-block begins. We can accomplish that by storing the new header in a temporary location. Moreover, each LN processing element will keep a counter to decide when it is time to produce and output the new header. Each counter of the LN processing element PE_i will be initialized to (i - 1)!. It is decremented in each time step. When the counter reaches 2N, it is time to produce the new header. When the counter reaches zero, it is time to output the new header.

Now, we have an algorithm to produce permutations for the LN processing elements. All we need is another algorithm to produce permutations for the SN processing elements.

Generate the permutations within a s-block

Akl's paper presented two ways to generate permutations for the SN processing elements. We have covered the first algorithm in section 5. 4. 1. It generates all the m-permutations of N objects with m processing elements in O(log m) running time. The problem with this algorithm is it has space complexity of O(m) and the O(log N) delay time will prevent the systolic algorithm from achieving optimal cost. The second algorithm generates permutations for the SN processing elements with constant delay time. It is as follows.

This algorithm generates permutations for the SN processing elements. In other words, it generates all of the k-block, $k < s$, assuming the first permutation of the k-block is known. We will also assume that k is bigger than 4. If $k \leq 4$, we will produce all the 4-permutations with a straightforward sequential algorithm.

Whenever a new k-block begins, our algorithm will produce D.k (k - 1)! times. Moreover, it will need to compute the next D.k within the (k - 1)! steps. The only exception is the last k-block of a k-run. In that case, the new permutation is produced by the algorithm with the LN processing elements. Therefore, each processing element PE_i, $4 < i \leq k$, will need two counters. One counter will count the number of objects produced within a k-block. We will call it the repetition counter (rc) where in each step rc = (rc + 1) mod (k-1)!. The counter rc is initialized to 1 when the algorithm begins. As a result, after each k-block, rc will return to 1. The other counter will count the number of blocks within a k-run. We will call it the block counter (bc). The counter bc is initialized to 1 when the algorithm begins and it is incremented every time rc returns to 1. Therefore, the first and the last k-block of a k-run are recognized when bc = 1 and bc = 0, respectively.

At the time a k-block begins, processing element PE_k is said to be the leader of permutation D.[k ... 1]. The permutation D.[k ... 1] should be stored, modified to the new header of the next k-block D'.[k ... 1] before the next k-block begins. At the same time, it is also the beginning of a (k-1)-block, (k-2)-block, . . . , etc. Therefore, the permutations D.[k-1 ... 1], D.[k-2 ... 1], ..., D.[4 ... 1] should be modified to the new header of their corresponding blocks as well. For example, assume a 7-block begins with D.[7 ... 1] = 8235679. It is also the beginning of a 6-block with D.[6 ... 1] = 235679, the beginning of a 5-block with D.[5 ... 1] = 35679 and the beginning of a 4-block with D.[4 ... 1] = 5679. Therefore, they should be modified to D'.[7 ... 1] = 9235679, D'.[6 ... 1] = 325679, D'.[5 ... 1] = 53679 and D'.[4 ... 1] = 6379. The modifications are done simultaneously and

stored separately. However, the replacement of the permutations with the modified values are done in different times since the block sizes of the different processing elements are different.

We will now explain how the D.[k ... 1]'s will be modified. First, note that when a k-block begins, D.k-1 < D.k-2 < ... < D.1. Therefore, to modify the permutation D.[i ... 1], i < k, all we need is to exchange D.i with D.i-1. To modify D.[k ... 1], we will employ a similar strategy as the way we update the LN processing elements. We will sweep the values of D.[k ... 1] across processing elements PE_j, $i \leq k$, for the first value that is bigger than D.k. Then, D.k is exchanged with this value and the rest of the permutation will be updated as the data sweep back across the processing elements toward PE_1. This will take at most 2k time steps. It has to be done before the current k-block expires. Therefore, 2k has to be smaller than (k-1)!. Since we have assumed that k > 4, therefore, $2k \leq (k - 1)!$ and the permutations will be updated in time.

We have noted that the modified permutations must be stored temporarily so that each processing element will be able to continue producing the old value until the next k-block begins. Next, we will discuss briefly how the modified permutations will be stored so the permutations will be distributed among processing elements instead of overloading one processing element. Each processing element will maintain a stack of k elements. These stacks will be connected virtually as a single stack, let's call it Q. The stack Q.[1 ... k * N] is actually represented by the stacks Q_i.[1 ... k] on each processing element PE_i. Therefore, Q.[x] is really $Q_{x \text{ div } k}$.[x mod k]. As the new header being produced by processing element PE_k, it is shifted left onto Q. If we try this on our previous example, Q = 8235679|325679|52678|6279. When the new headers are needed, they are shifted right to the corresponding processing elements and output. The implementation details are presented in Akl's paper [Akl & Meuer [3]] and will be omitted here.

That concludes the description of this algorithm. It is obvious that each processing element will produce one position of the permutation at each time step. Therefore, the overall running time is O(N!). At each time step, the duty of each processing element is to receive some data from its neighbor, modify the data if necessary and shift the data left or right. From the above description of the algorithm, clearly each step can be done in constant time. Therefore, the delay time between successive permutations is O(1). The systolic array consists of N processing elements. Therefore, the total computation cost is O(N!) * N * O(1) = O(N! * N). This matches the optimal sequential computation cost. Furthermore, each processing element maintains at most O(s-1) space. And $(s - 1)! \geq 2N$. The value s grows very slowly as N increases. Therefore, the memory requirement for each processing element is quite minimal.

The problem with this algorithm is the computations performed on each processing element are not very regular. There are basically three sets of computations: generating permutations for LN processing elements, generating permutation within a k-blocks with $4 \leq k \leq s$, and generating permutations of 4 objects. Moreover, it is obvious that the SN processing elements are much busier than the LN processing elements. A systolic array requires the computations performed on its processing elements to be regular so that data can travel through the systolic array in a lockstep and rhythmic fashion. Even though each processing element in this algorithm finishes its time step before its successor does and therefore maintains the continuity of the algorithm, it will be complicated to build a systolic array for this algorithm since the functions of the processing elements differ.

Finally, this algorithm only generates full permutations of N objects. Sometimes, it is beneficial to generate the m-permutations of N objects. In the next section, we will design a new algorithm that will generate all the m-permutations of N objects in lexicographic order with similar complexity but more regular structure.

5. 4. 4 A New Systolic Algorithm

In this section, we will design a new systolic algorithm that generates all the m-permutations of N objects in lexicographic order. Our algorithm will use a typical linear array of m processing elements. Each processing element PE_i, $1 \le i \le m$, will be responsible for generating the ith position of a permutation indexing from left to right. Our algorithm is based on the idea that each processing element will produce the same object ${}^{(N-i)}P_{(m-i)}$ times. What we need is to develop a way to compute the next object within the time frame. To better understand our algorithm, Figure 5.5 illustrate the states of our systolic array and the first 10 permutations it produces assuming we are generation 9-permutations of 12 objects.

Each processing element maintains a queue Q_i. $Q_1[1 \ldots N] = \{1,2,3, \ldots, N\}$ and $Q_i[1 \ldots N] = Q_{i-1}[i \ldots N]$, $2 \le i \le m$. We can see from Figure 5.5 that each processing element PE_i will produce the same object from Q_i in ascending order for ${}^{(N-i)}P_{(m-i)}$ times. After the processing elements have gone through all the objects in their corresponding queues, they need new queues of objects. From Figure 5.5, we can see that after PE_9 generated objects 9, 10, 11 and 12, the queue should read 8, 10. 11 and 12. In order for the processing elements to continue to produce the objects without delay, the new queues must be generated before the processing elements have gone through all the objects in their queues. Using the notation from the previous section, we will define a k-block as the set of permutations where the ith position, $i \le k$, does not change. And a k-run is a k+1 block. After processing element PE_k starts a k-block, it must generate a new queue within ${}^{(N-i)}P_{(m-i)}$ time steps. Therefore, our goal is to develop an algorithm that will produce a queue Q_i for processing element PE_i in ${}^{(N-i)}P_{(m-i)}$ time steps.

Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
1								
2	2							
3	3	3						
4	4	4	4					
5	5	5	5	5				
6	6	6	6	6	6			
7	7	7	7	7	7	7		
8	8	8	8	8	8	8	8	
9	9	9	9	9	9	9	9	9
10	10	10	10	10	10	10	10	10
11	11	11	11	11	11	11	11	11
12	12	12	12	12	12	12	12	12
PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	PE_7	PE_8	PE_9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	10
1	2	3	4	5	6	7	8	11
1	2	3	4	5	6	7	8	12
1	2	3	4	5	6	7	9	8
1	2	3	4	5	6	7	9	10
1	2	3	4	5	6	7	9	11
1	2	3	4	5	6	7	9	12
1	2	3	4	5	6	7	10	8
1	2	3	4	5	6	7	10	9

Figure 5. 5: State of the systolic array during generation of the first 10 9-permutations of 12 objects.

We have already stated that each processing element PE_i will maintain a queue Q_i. At the beginning of the algorithm, each queue Q_i is initialized to be i ... N. At each time step, PE_i will output an object from Q_i. The objects are chosen by picking each object in Q_i ${}^{(N-i)}P_{(m-i)}$ times. Therefore, each PE_i needs a repetition counter rc_i to count the repetition of the current chosen object in Q_i, i.e. at each time step, $rc_i = (rc_i + 1) \bmod {}^{(N-i)}P_{(m-i)}$. The counter rc_i is initialized to 1 at the beginning of each i-block.

Each PE_i also needs a pointer ptr_i to store the position of the current object from Q_i. At the beginning of an i-run, ptr_i points to $Q_i[1]$. Every time a new i-block begins, ptr_i is incremented, i.e. every time $rc_i = 1$, $ptr_i = (ptr_i + 1) \bmod (N - i)$. When $ptr_i = 0$, it signifies the end of an i-run and the beginning of producing outputs with a new list of objects. When $ptr_1 = 0$, it means the end of the algorithm.

Each PE_i is responsible for generating the new list of objects for PE_{i+1}. We can see from Figure 5.5 that the new list of objects of PE_{i+1} is actually the list of objects from PE_i except the object going to be outputted by PE_i in the next i-block, i.e. $ptr_i + 1$. Therefore, at each time step, PE_i will send one object from its queue in ascending order to PE_{i+1} while skipping the object pointed by ptr_i. We will use another pointer $update\_ptr_i$ to point to the current object that is sent to PE_{i+1}, i.e. $update\_ptr_i = (update\_ptr_i + 1) \bmod (N - i)$ at each time step and it is initialized to 1 at the beginning of each i-block.

We will define a function add_queue(x, q) which will add an object x to the tail of the queue q. As a result, at the beginning of each i-run except the very first one, the queue of PE_i will consist of two lists of objects - the old objects list and the new objects list. Therefore, before each PE_i starts producing output from the queue, it needs to remove the old list from the queue. We will define a function delete_queue to do that, delete_queue(n, q) will delete n objects from the head of the queue q.

The systolic array transmits data with communication links. Using the notations from section 4. 3. 4, we define a d_link as the communication link connecting two processing element. Moreover, d_in and d_out are the input and output data on the d_link, respectively. We also define a c_link as the output link for each processing elements. Thus, resulting in the architecture of Figure 5.6.

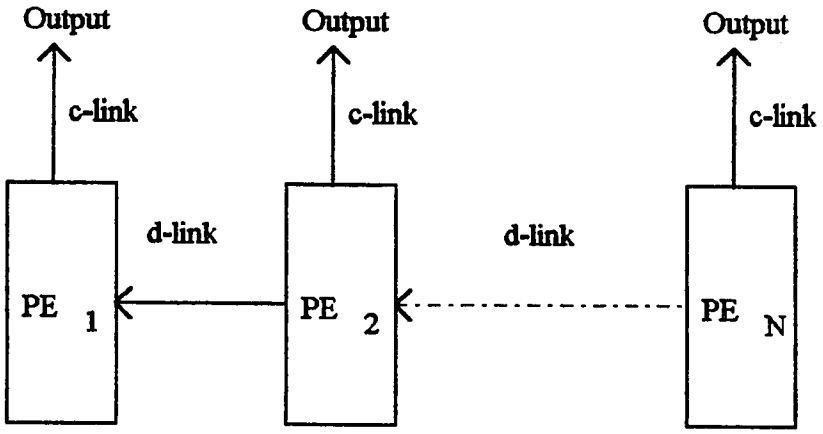


Figure 5.6: Systolic array used in Algorithm 5.6.

Now, we present the complete algorithm as Algorithm 5.6.

Algorithm 5. 6 A new systolic algorithm for generating m-permutations of N objects

```
New_systolic_permutation(m, N )
{ int Q_i[1. . . . 2N];
   int ptr_i;
   int update_ptr_i;
   int rc_i;
   int rep_i, i;
   for (i = 1; i ≤ m; i++)
    for each processing element PE_i do in parallel {
     /* Initialization phrase */
      rc_i = ptr_i = 0
      Q_i = [i . . .  N];
      /* So we don't need to compute this factorial function every time.  */
      rep_i = N-iP_m-i;
```

```
/* Execution phrase */
While (ptr_1 ≠ 0) {
  /* The beginning of a i-block. */
  if (rc_i == 0) {
     rc_i = update_ptr_i = 1;
     ptr_i = (ptr_i + 1) mod (N - i );
  }
  /* Add the new object to the queue. */
  add_queue(d_in, Q_i);
  /* The beginning of an i-run. */
  if (ptr_i = 0) {
    delele_queue(N-i, Q_i);
    ptr_i = update_ptr_i = 1;
  }
  c_out = Q_i[ptr_i];
  /* Sending object to PE_{i+1} while skipping the object pointed by ptr_i. */
  if ( (ptr_i + 1) == update_ptr_i )
     update_ptr_i = update_ptr_i + 1;
  d_out = Q_i[update_ptr_i];
  update_ptr_i = (update_ptr_i + 1) mod (N - i);
  rc_i = rc_i + 1 mod rep_i;
 } /* End while. */
} /* End for */
} /* End procedure new_systolic_permutation. */
```

Next, we will prove the correctness of our algorithm.

First, we will prove a new list of objects will be generated in time before the next i-block. Each i-block consists of ${}^{N-i}P_{m-i}$ permutations, therefore, the time it takes to generate a new list of objects must be less than ${}^{N-i}P_{m-i}$ time steps. A new list of objects consists of (N-i) objects. An object is sent to PE_{i+1} in each time step, therefore, the time it takes to generate a new list of objects is N-1 time steps. Thus, we need to prove that $N - i \leq {}^{N-i}P_{m-i}$. Recall that ${}^{N-i}P_{m-i} = (N - i)! / (N - i - (m - i))!$. Therefore, we need to prove

$$N - i \leq (N - i)! / (N - i - (m - i))! \text{ , or}$$
$$N - i \leq (N - i) * (N - i - 1)! / (N - i - (m - i))!$$

Equivalently, it suffices to prove

$$(N - i - 1)! / (N - i - (m - i))! \geq 1, \text{ i. e. ,}$$
$$N - i - 1 \geq N - m \text{ , i. e. ,}$$
$$m \geq i + 1$$

The last processing element PE_m does not need to generate an objects list, therefore, $i \leq m - 1$. Thus, $m \geq i + 1$. This proves that the new list of objects will always be generated in time.

Next, we will prove that our algorithm will generate all the permutations in lexicographic order. We do that by proving if a permutation P_x is generated after a permutation P_y, then $P_x < P_y$ in lexicographic order. Recall that for two permutations

$P_x = \{x_1, x_2, ..., x_m\}$ and $P_y = \{y_1, y_2, ..., y_m\}$, P_x precedes P_y in lexicographic order if there exists an i, $1 \le i \le m$, such that $x_j = y_j$ for all $j < i$ and $x_i < y_i$. Let's call this i the transition element.

We will prove this by induction. The first permutation generated by this algorithm is (1 2 3 ... m-1 m) with $Q_m = \{m, m+1, m+2, ..., N\}$ and $Q_{m-1} = \{m-1, m, m+1, ..., N\}$. The second permutation generated by this algorithm is (1 2 3 ... p_{m-1} p_m) where p_{m-1} is either m-1 or m and p_m is either m+1 or m-1, respectively. Clearly, in the first case the transition element is p_{m-1} and in the second case it is p_m. Therefore, our algorithm works for the first two permutations and proves our base case.

Now, we will assume that our algorithm works for the kth permutation. This means for permutations P_k and P_{k+1}, there exists a transition element p_i. All we need to do is to prove a transition element exists for the (k+1)th permutation.

Let $P_{k+1} = (p_1 p_2 ... p_i ... p_m)$ and $P_{k+2} = (p'_1 p'_2 ... p'_i ... p'_m)$. From the above assumption, p_i is the transition element in P_k. This means P_{k+1} is the beginning of an i-block. This also means P_{k+1} is the beginning of an (i+1)-block, (i+2)-block, ... , m-block. The transition element in P_{k+1} is the smallest indexed object which get incremented in P_{k+2}. Each processing element PE_j, $j \ge i$, will produce the same object ${}^{N-j}P_{m-j}$ times. Therefore, the object get incremented in P_{k+2} is the object at processing element PE_j where ${}^{N-j}P_{m-j} = 1$. If $m < N$, ${}^{N-j}P_{m-j} = 1$ when j is m, thus p_m is the transition element. If $m = N$, ${}^{N-j}P_{m-j} = 1$ when j is either (m-1) or m, thus p_{m-1} is the transition element. In either case , PE_j will produce the next object from Q_j that is following p_j. And all PE_k, $k < j$, will produce the same objects. The fact that our queues are always in ascending order means that the next object p'_j will be bigger than p_j. Therefore, a transition element p_j exists for P_{k+1}, and proves that P_{k+1} precedes P_{k+2} in lexicographic order.

Since both the base case and the inductive case are true, our claim that for all successive permutations P_x and P_y generated by our algorithm, $P_x < P_y$, follows by induction. Since our algorithm will only terminate when PE_1 has gone through all the N objects in its list (${}^{N-1}P_{m-1}$) times, our algorithm will generate $N * {}^{N-1}P_{m-1} = {}^{N}P_{m}$ permutations. Therefore, our algorithm generates all the m-permutations of N objects in lexicographic order.

Finally, we will analyze the complexity of our algorithm. First, each processing element produces one position of a m-permutation in each time step. Since they are operating in parallel, one permutation will be generated in each time step. There are ${}^{N}P_{m}$ permutations, therefore, each processing element will operate for ${}^{N}P_{m}$ time steps. In each time step, all the processing elements need to do are receive an object, output an object and send an object. Clearly, these are all constant time operations. Therefore, each time step has constant running time. It also implies that the delay time between successive permutations is constant.

Since each processing element performs ${}^{N}P_{m}$ time steps and each time step has constant running time, the overall running time for each processing element is $O({}^{N}P_{m})$.

There are m processing elements, therefore, the overall computational cost is $O({}^{N}P_{m} * m)$. This matches the optimal sequential time and thus our algorithm is cost optimal.

Each processing element needs to maintain a queue consists of two lists of objects. Each list has size at most N, therefore, the space requirement for each processing element is O(2N). Actually, the queue Q_i for processing element PE_i has size N - i + 1. The overall space requirement for the queue is 2 * (N + N-1 + N-2 + ... + N-m+1). Therefore, the overall space requirement for the queue is

$$2 * (N * (N +1) /2 - (N - m) * (N - m + 1) / 2 = O(N^2).$$

The difference between our algorithm with Akl's systolic algorithm is our algorithm has a more regular structure. Each processing element performs essentially the same functions.

The time complexities of both algorithms are quite similar. However, Akl's algorithm does not require the queues and therefore has a more constant space requirement. On the other hand, the space requirement of our algorithm grows with the size of the problem.

5. 4. 5 Adaptive Systolic Algorithm

Our algorithm can be modified to run with an arbitrary number of processing elements. If the number of processing elements k available is fewer than m, all we need to do is distribute the work load among the available processing elements. In other words, processing element PE_i, $1 \le i \le k$, will act as processing elements $(i - 1) * \lceil m / k \rceil + 1$ to $(i - 1) * \lceil m / k \rceil + \lceil m / k \rceil$, and Algorithm 5.6 will be modified to Algorithm 5.7.

There are three modifications in Algorithm 5.7. At Modification #1, processing element PE_j, $1 \le j \le k$, will act as processing elements $(j - 1) * \lceil m / k \rceil + 1$ to $(j - 1) * \lceil m / k \rceil + \lceil m / k \rceil$ in each time step. Therefore, each time step will consist of $\lceil m / k \rceil$ iterations with j being incremented at each iteration. Moreover, updating the queues within the same processing element does not require sending the values through the communication links. Therefore, at Modification #2 and #3, each processing element will only receive input or send output through the communication links at the beginning or the ending of the loop, respectively.

Algorithm 5.7 Adaptive systolic algorithm which uses k processing elements, $k \leq m$.

```
Adaptive_systolic_permutation(m, N, k )
{ int Q_i[1. . . . 2N];
  int ptr_i;
  int update_ptr_i;
  int rc_i;
  int rep_i, i;
  /* Modification #1 */
  for ( j = 1; j ≤ k; j++)
   for each processing element PE_j do in parallel
    head = ( j - 1) * ⌈ m / k ⌉+ 1;
    tail = ( j - 1) * ⌈ m / k ⌉+⌈ m / k ⌉;
    for ( i = head;  i ≤ tail ;  i++) {
    /* Initialization phase */
     rc_i = ptr_i = 0
     Q_i = [i . . .  N];
     /* So we don't need to compute this factorial function every time.  */
     rep_i = N-iP_m-i;
  /* Execution phase */
    while (ptr_1 ≠  0) {
    /* The beginning of a i-block.  */
     if (rc_i == 0) {
        rc_i = update_ptr_i = 1;
        ptr_i = (ptr_i + 1) mod (N - i );  }
```

```
/* Add the new object to the queue.  */
    /* Modification #2 */
    if (i == head)
      add_queue(d_in, Qi)
    else
      add_queue(next_input, Qi);
    /* The beginning of an i-run.  */
    if (ptr_i = 0) {
      delele_queue(N-i, Q_i);
       ptr_i = update_ptr_i = 1; }
    c_out = Q_i[ptr_i];
    /* Sending object to PE_{i+1} while skipping the object pointed by ptr_i.  */
    if ( (ptr_i + 1) == update_ptr_i )
       update_ptr_i = update_ptr_i + 1;
    /* Modification #3 */
    if (i == tail)
      d_out = Q_i[update_ptr_i] .
    else
      next_input = Q_i[update_ptr_i];
    update_ptr_i = (update_ptr_i + 1) mod (N - i);
     rc_i = rc_i + 1 mod rep_i;
   } /* End while.  */
  } /* End for */
} /* End procedure Adaptive_systolic_permutation.  */
```

Since now each time step consists of $\lceil m / k \rceil$ iterations, the delay time between successive processing elements increases from O(1) to $O(\lceil m / k \rceil)$. Therefore, implementing our algorithm on a system with k processing elements, $k \leq m$, results in a slow down of $\lceil m / k \rceil$.

The modification is so simple because our algorithm is very regular. Each processing element performs equivalent operations. This is another advantage of our algorithm comparing with other algorithms such as Akl's. In Akl's systolic algorithm, each processing element can has different functionalities, especially between the large numbered processing elements and the small numbered processing elements. If we partition the processing elements in Akl's algorithm the same way as we partition the processing elements in our algorithm, then a processing element may take up the responsibility of both a large numbered processing element and a small numbered processing element. As a result, the systolic algorithm for each processing element will require a much larger scale modification.

The last thing we need to point out about our adaptive algorithm is the choice of using the ceiling function. Partitioning the processing element using the ceiling function does not distribute the work loads as evenly as possible. Fortunately, it does not affect the efficiency of our algorithm. For example, suppose we are distributing the work loads of six processing elements among five processing elements, i.e. $m = 6$ and $k = 5$. Using our function, the distribution is represented as Figure 5.7 a. However, a more even distribution of work loads is represented as Figure 5.7 b.

PE:	1	2	3	4	5
Act as PE:	1, 2	3, 4	5, 6	idle	idle

Figure 5.7a: Distribution of work loads of 6 PEs among 5 PEs.

PE:	1	2	3	4	5
Act as PE:	1, 2	3	4	5	6

Figure 5.7 b: More even distribution of work loads of 6 PEs among 5 PEs.

Fortunately, in both cases the delay time between successive processing elements is still $\lceil m / k \rceil$. In Figure 5.7 a, all the processing elements have time steps with the same complexity, $O(\lceil m / k \rceil)$. On the other hand, in Figure 5.7 b, processing elements PE_1 has complexity $O(\lceil m / k \rceil)$ but the other processing elements have complexity O(1). However, since PE_2 cannot start execution until PE_1 is finished, PE_2 will have to wait for PE_1 to finish and thus increases its complexity to $O(\lceil m / k \rceil)$. And the same situation happens at PE_3, PE_4, ... , etc. As a result, all the processing elements will have complexity $O(\lceil m / k \rceil)$ despite the more even distribution of work loads.

Therefore, our adaptive algorithm has running time $O({}^{N}P_m * \lceil m / k \rceil)$. The total cost remains the same - $O({}^{N}P_m * \lceil m / k \rceil * k) = O({}^{N}P_m * m)$. The delay time is $O(\lceil m / k \rceil)$. In the case that k = m as in our original algorithm, the delay time becomes O(1).

Sections:	# processors	Running time	Total cost	Delay time	Space
5.3.1	1	$O({}^{N}P_{m} * m)$	$O({}^{N}P_{m} * m)$	O(m)	O(m)
5.3.2	1	$O({}^{N}P_{m} * m)$	$O({}^{N}P_{m} * m)$	O(m)	O(m)
5.4.1	m	$O({}^{N}P_{m} * \log m)$	$O({}^{N}P_{m} * \log m * m)$	O(log m)	O(m)
5.4.2	variable - k	$O({}^{N}P_{m}/k * m)$	$O({}^{N}P_{m} * m)$	O(m)	O(m)
5.4.3	N	O(N!)	O(N! * N)	O(1)	O(1)
5.4.4	m	$O({}^{N}P_{m})$	$O({}^{N}P_{m} * m)$	O(1)	O(N) * m
5.4.5	variable - k	$O({}^{N}P_{m}) * \lceil m / k \rceil$	$O({}^{N}P_{m} * m)$	$O(\lceil m / k \rceil)$	O(N) * m

Figure 5.8: Summary of all the lexicographic algorithms presented in this chapter.

5. 5 Summary

In this thesis, we have presented six algorithms for generating permutations in lexicographic order. Figure 5.8 listed all the algorithms presented in this chapter by their section numbers and their complexities.

The first algorithm is a straight forward sequential approach to compute the successive permutation from an existing one. The second algorithm is a simpler sequential approach to achieve the same thing. It sweeps the objects across the elements of a permutation, searches for an element to update and updates the elements to its right. It is essentially the same strategy Akl used in his systolic algorithm to generate s run for the LN processing elements.

The first parallel algorithm is a parallelized version of the first sequential algorithm. It parallelizes the steps to compute each successive permutation from an existing one. The second parallel algorithm using enumeration of permutation takes a different approach. Instead of parallelizing the process of generating each permutation, it parallelizes the steps for generating all the permutations. Each permutation is still being generated sequentially. However, multiple permutations are generated at the same time.

Akl's systolic algorithm partitions the generation of each permutation into three parts, the LN processing elements, the SN processing elements, and processing elements PE_i, $i \leq 4$. The algorithm utilizes the idea of an i-block and an i-run. It parallelizes the steps to generate each i-run, with computation for different i carried out at the same time.

Our algorithm matches or exceeds all of the other algorithms in term of efficiency. It is running time optimal, cost optimal with constant delay time and variable m. The only problem is our algorithm has linear space requirement. This is a problem especially for

a systolic algorithm because it seems that a systolic array is amenable to VLSI implementation if the space requirement for each processing element is independent of the problem size. An open question left by this thesis is to enhance our algorithm to reduce the space requirement to constant time. It can probably be accomplished by distributing the queues across the processing elements.

Finally, I hope this thesis will not only illustrate how we can handle the problem of generating permutations, but also demonstrate that we can parallelize a problem with different approaches while showing the advantages and drawbacks of each of the approaches.

Bibliography

1) Ajtai, M. and Komlos, J. and Szemeredi, E., "An O(n log n) sorting network," *In Proceeding of the 15th Annual ACM Symposium on Theory of Computing*, pp.1-9, April 1983.

2) Akl, Selim G., *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, NJ., 1989.

3) Akl, Selim G. and Meuer, Henk, "An Optimal Systolic Algorithm for Generating Permutations in Lexicographic Order," *Journal of Parallel and Distributed Computing*, vol. 20, no. 1, pp.84-91, January 1994.

4) Almasi, George and Gottlieb, Allan, *Highly Parallel Computing*, The Benjamin/Cumming Publishing Company, Inc., Redwood City, CA., 1989.

5) Chen, G. H. and Chern, Maw-Sheng, "Parallel Generation of Permutations and Combinations," *Bit Magazine*, vol. 26, no. 3, pp.277-283, December 1986.

6) Cole, Richard, "Parallel Merge Sort," *Synthesis of Parallel Algorithms*, pp.453 Morgan Kaufmann Publishers, Inc. San Mateo, CA., 1993.

7) Dijkstra, E. W., *A discipline of programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

8) Evans, D. J., "A Parallel Sorting - Merging Algorithm for Tightly Coupled Multiprocessors," *Parallel Computing*, vol. 14, no. 1, pp.111, May 1990.

9) Heap, B. R., "Permutations by Interchanges," *Computer Journal*, vol. 6, pp.293-294, April 1963.

10) Kuman, V. and Grama, A. and Gupta, A. and Karypis, G., *Introduction to Parallel Computing: Design and Analysis of Algorithms*, The Benjamin/Cumming Publishing Company, Inc., Redwood City, CA., 1994.

11) Leighton, Ed Thomas, *Introduction to Parallel Algorithm and Architectures : Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA., 1992.

12) Lin, Chau-Jy, "Parallel Generation of Permutations on Systolic Arrays," *Parallel Computing*, vol. 15, pp.267-276, September 1990.

13) Ord-Smith, R. J., "Generation of Permutations in Lexicographical Order," *Communications of the ACM*, vol. 11, no. 2, pp.117, February 1968.

14) Quinn, Michael J., *Parallel Computing : Theory and Practice*, McGraw-Hill, Inc., 1994.

15) Quinn, Michael J., "Parallel Sorting Algorithms for Tightly Coupled Multiprocessors," *Parallel Computing*, vol. 6, no. 3, pp.349, March 1988.

16) Sado, Kazuhiro and Igarashi Yoshihide, "Some Properties of the Parallel Bubbling and Parallel Sorts on a Mesh-connected Processor Array," *Discrete Algorithms and Complexity*, pp.161, Proceedings of the Japan-US Joint Seminar '86, Academic Press, Inc., Orlando, Fl., 1986.

17) Schmeichel, E. F., *Class Note: CS 255 Parallel Algorithms*, Fall 1992

18) Sedgewick, Robert, "Permutation Generation Methods," *Computer Surveys*, vol. 9, no. 2, pp.137-164, June 1977.

19) Shen, Mok-Kong, "Generation of Permutations in Lexicographical Order," *Communication of the ACM*, vol. 6, no. 8, pp.517, September 1963.

20) Shimrat, M. and Schrack G. F., "Permutation in Lexicographical Order," *Communication of the ACM*, vol. 5, no. 6, pp.346, June 1962.

21) Smith, Justin R., *The Design and Analysis of Parallel Algorithms*, Oxford University Press, New York, NY., 1993.

22) Tal, Doron and Navathe, Sham and Graham, Scott, "On Parallel Architecture," *A Post Conference Publication Based upon the Proceedings of PARABASE '90*, pp.10, IEEE Computer Society Press, Los Alamitos, CA, 1990.

23) Tompkins, C., "Machine Attacks on Problems Whose Variables are Permutations," *Numerical Analysis*, vol. 6, Proceedings Symposium in Applied Math., McGraw Hill Inc., NY., 1956.

24) Tsay, Jong-Chuang and Lee, Wei-Ping, "An Optimal Parallel Algorithm for Generating Permutations in Minimal Change Order," *Parallel Computing*, vol. 20, no. 3, pp.353-361, March 1994.

25) Wells, M. B., "Generation of Permutations by Transposition," *Math. Comp.* vol. 15, pp.192-195, August 1961.

26) Zaks, Shmuel 1984, "A New Algorithm for Generation of Permutations," *Bit Magazine*, vol. 24, no. 2, pp.196-204, September 1984.

Index